Studies on Design Automation and Arithmetic Circuit Design for Single-Flux-Quantum Digital Circuits

Koji Obata

Abstract

Superconductive single-flux-quantum (SFQ) circuit technology attracts attention as a next generation technology of integrated circuits because of its ultra-fast computation speed and low power consumption. In SFQ digital circuits, unlike CMOS digital circuits, a pulse is used as a carrier of information and the representation of the logic values is different from that in CMOS digital circuits. Therefore, design automation algorithms and structure of arithmetic circuits suitable for SFQ digital circuits are different from those for CMOS digital circuits. In addition, design of SFQ circuits has been carried out largely manually. For advancing studies of SFQ digital circuits, design automation algorithms which can design high-performance SFQ circuits are important. Furthermore, studies of circuit structure suitable for SFQ arithmetic circuits are also important for designing high-performance circuits. In this dissertation, several design automation algorithms and design of a multiplier which is one of the most important arithmetic circuits are proposed for SFQ digital circuits.

In Chapter 1, the background and the outline of the dissertation are described.

In Chapter 2, the basis of SFQ circuits and the representation of logic values for SFQ digital circuits are described. There are two methods of representation of the logic values. One is ‘dual-rail representation’ in which “1” and “0” lines are used and the other is ‘synchronous clocking representation’ in which synchronizing clocks are used.

In Chapter 3, a new method of logic synthesis for dual-rail SFQ digital circuits are proposed. For representing logic functions, a root-shared binary decision diagram (RSBDD) which is a directed acyclic graph constructed from binary decision diagrams is proposed. In the method, first an RSBDD is constructed from given logic functions, and then the number of nodes in the constructed RSBDD is reduced by variable re-ordering. Finally, a dual-rail SFQ digital circuit is synthesized from the reduced RSBDD. The experimental results on benchmark circuits show that the proposed method synthesizes dual-rail SFQ digital circuits that consist of about 27.1% fewer logic elements than those synthesized by a Transduction-based method on average.

In Chapter 4, an algorithm for clock scheduling of synchronous clocking SFQ digital circuits is proposed. In synchronous clocking SFQ digital circuits, all logic gates are driven by clock pulses. Appropriate clock scheduling makes clock frequency of the circuits higher. Given a clock period, the proposed algorithm determines the arrival time of clock pulses of each gate and the delay that should be inserted. The experimental results on benchmark circuits show that inserted delay elements by the proposed algorithm are 59.0% fewer and the height of clock trees are 40.4% shorter on average than those by a straightforward algorithm. The proposed algorithm can also be used to minimize the clock period, thus obtaining 19.0% shorter clock periods on average.

In Chapter 5, a synthesis method of sequential circuits is proposed for synchronous clocking SFQ digital circuits. Since all logic gates of synchronous clocking SFQ digital circuits are driven by a clock signal, synthesis methods of sequential circuits for CMOS digital circuits cannot derive the full power of high-throughput computation of SFQ circuit technology. In the method, a ‘state module’ consisting of a D flip-flop (DFF) and several AND gates is used. First, states of a sequential machine are encoded by one-hot encoding and state modules are assigned to the states one by one, and then, the modules are connected with each other according to the state transition. For the connection, confluence buffers (CBs), i.e., merger gates without clock signals are used. Consequently, gates driven by a clock signal are removed from its feedback loops, and therefore, a high-throughput SFQ sequential circuit is achieved. The experimental results on benchmark circuits show that compared with a conventional method for CMOS digital circuits, the proposed method synthesizes circuits that work with 4.9 times higher clock frequency and have 17.3% more gates on average.

In Chapter 6, an integer multiplier with systolic array structure is proposed for synchronous clocking SFQ digital circuits. The systolic array is a circuit structure for VLSIs and consists of regularly arranged simple processing elements (PEs). For evaluating the proposed multiplier, a 4-bit systolic multiplier and a 4-bit array multiplier which is one of the most typical parallel multipliers are designed and compared with each other. The results of the design and a digital simulation show that the circuit area of the 4-bit systolic multiplier is almost the half of that of the 4-bit array multiplier, and the latency is about 1.5 times longer. Our estimation of the performance of larger scale multipliers shows that the proposed systolic multiplier achieves comparable latency to the array multiplier with extremely smaller circuit area when the bit-width of input is large. A 1-bit PE of the systolic multiplier is fabricated using NEC standard Nb process and successfully tested at low speed.

In Chapter 7, conclusion and future works are stated. The knowledge obtained through the design automation algorithms will be bases of the development of computer-aided design (CAD) systems for SFQ digital circuits. The result obtained through the study of the systolic multiplier is valuable knowledge for designing SFQ arithmetic circuits. Development of SFQ-specific algorithms and methods makes the performance of SFQ digital circuits higher.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Recent advances in information and communication technology have been supported by the continuous progress of CMOS integrated circuits. However, the progress faces difficulties because of the limit of miniaturization, the heat dissipation, the increase of interconnect delay and so on. To overcome the difficulties, many technologies have been studied. Superconductive single-flux-quantum (SFQ) circuit technology[1] is one of such technologies. In the report of International Technology Roadmap for Semiconductor (ITRS) published in 2003, SFQ circuit technology was ranked at the highest position for the next generation technology of integrated circuits[2]. Various studies on SFQ circuit technology have been carried out[3–24].

SFQ circuits consist of Josephson junctions (JJs) and inductances. An SFQ pulse which is generated at a JJ is used as a carrier of information. The width of an SFQ pulse is several picoseconds and the height is about 1 mV. Switching energy of a gate of SFQ circuits is much smaller than that of CMOS circuits and switching speed is faster. By using SFQ circuits, ultra-fast computation unachievable by CMOS circuits is expected. Until now, process technologies[4], cell libraries[5, 6], processor architectures[7–12], network switches[13], arithmetic circuits[14, 15], interconnection technologies[16–18], cryocooled systems[19] and so on have been studied using tiny prototype circuits. In addition, design of the circuits has been carried out largely by manually placing and routing optimized parts called cells. It is expected that relatively large integrated circuits will be achieved in the near future by the advance of process technology. To design larger scale circuits, computer-aided design (CAD) including design automation is indispensable and studies on design automation algorithms suitable for SFQ circuits are important.

In SFQ digital circuits, pulses are used for representing the logic values “1” and “0.” When the presence and the absence of a pulse are simply assigned to the logic values “1” and “0,” the logic value “0” and the state of “no signal” is indistinguishable. Therefore, in SFQ digital circuits, two methods are used for representing the logic values[3]. One is ‘dual-rail representation’ in which “1” and “0” lines are used and the other is ‘synchronous clocking representation’ in which synchronizing clocks are used. Since these representation methods are different from those of CMOS digital circuits, circuit structure suitable for arithmetic circuits is different.

As interconnections in SFQ digital circuits, Josephson-transmission-lines (JTLs)[1] have been used. JTLs consist of JJs and the transmission delay of unit length is comparable with that of logic gates. Now, transmission lines without JJs are developed as next generation interconnections of SFQ circuits[16–18]. The transmission lines are called passive-transmission-lines (PTLs) and can transmit pulses at almost the speed of light. Since many studies have been carried out on designs using JTLs, the potential of PTLs is not derived.

In this dissertation, we propose design automation algorithms and design of an arithmetic circuit for SFQ digital circuits. Advancement of SFQ circuit technology largely relies on the development of CAD systems which can achieve high-performance (e.g., small-area, high-throughput) circuits and can design circuits systematically. There are many study topics such as logic synthesis, clock scheduling, clock tree synthesis, placement and routing and verification. In addition, studies on circuit structure of arithmetic circuits suitable for SFQ digital circuits are also important for the advancement of SFQ circuit technology because suitable circuit structure of SFQ arithmetic circuits is different from that of CMOS arithmetic circuits. Several studies have been carried out[20, 23, 24].

For design automation algorithms, we focus on logic synthesis and clock scheduling. For dual-rail SFQ digital circuits, we propose a new method of logic synthesis. Although dual-rail representation has some advantages in comparison with synchronous clocking representation, it has not been used because circuit area tends to be larger. It is important to develop a method of logic synthesis which can reduce circuit area. For synchronous clocking SFQ digital circuits with PTLs, we propose an algorithm of clock scheduling. Clock scheduling becomes important when PTLs are used as interconnections because it largely affects the performance of circuits. For synchronous clocking SFQ digital circuits, we propose a synthesis method of sequential circuits. Since all logic gates are driven by clock pulses, conventional synthesis methods of sequential circuits spoil the power of high-throughput computation of an SFQ digital circuit. It is important to develop a new method which can utilize the power. As an SFQ arithmetic circuit, we focus on multiplication and propose a multiplier suitable for SFQ digital circuits because multiplication is one of the most important arithmetic operations.

Using the design automation algorithms to be proposed, we can design high-performance SFQ digital circuits systematically. The knowledge obtained through the studies about design automation algorithms will be bases of the development of CAD systems for SFQ digital circuits. In addition, the result obtained through the study of the multiplier is valuable knowledge to design SFQ arithmetic circuits. Development of SFQ-specific algorithms and methods makes the performance of SFQ digital circuits higher.

1.2 Outline of the Dissertation

In Chapter 2, we describe the basis of SFQ circuits and the representation of logic values for SFQ digital circuits.

In Chapter 3, we propose a method of logic synthesis for dual-rail SFQ digital circuits.

Dual-rail representation has some advantages in comparison with synchronous clocking representation. However, since circuit area using dual-rail representation usually becomes larger, dual-rail representation has not been used. If small-area can be achieved using dual-rail representation, adoption of dual-rail representation becomes a promising approach for designing SFQ digital circuits. The experimental results on benchmark circuits show that the proposed method can synthesize dual-rail SFQ digital circuits which consist of about 27.1% fewer logic elements than those synthesized by a Transduction-based method on average.

In Chapter 4, we propose a clock scheduling algorithm for synchronous clocking SFQ digital circuits with PTLs. Clock scheduling becomes important when PTLs are used as interconnections because it largely affects the performance (e.g., clock frequency and/or area) of circuits. Development of good clock scheduling algorithms leads synchronous clocking SFQ digital circuits to high-throughput and small-area circuits. The experimental results on benchmark circuits show that for a given clock period, the proposed algorithm can obtain near optimal solutions in which inserted delay elements are 59.0% fewer and the height of clock trees are 40.4% shorter on average than those by a straightforward algorithm. The minimum clock periods are 19.0% shorter on average than those by the straightforward algorithm.

In Chapter 5, we propose a synthesis method of sequential circuits (circuits with feedback loops) for synchronous clocking SFQ digital circuits. Since all logic gates are driven by clock pulses, conventional synthesis methods of sequential circuits spoil the power of high-throughput computation of an SFQ digital circuit. For achieving high-performance SFQ sequential circuits, it is important to develop a new method which can utilize the power. The experimental results on benchmark circuits show that compared with a conventional method for CMOS digital circuits, the proposed method synthesizes circuits that work with 4.9 times higher clock frequency and have 17.3% more gates on average.

In Chapter 6, we propose an integer multiplier suitable for synchronous clocking SFQ digital circuits based on the systolic array scheme. Since SFQ digital circuits work by pulse logic, logic gates of SFQ digital circuits have different features from those of CMOS digital circuits. Suitable circuit structure of SFQ arithmetic circuits is different from that of CMOS arithmetic circuits. Therefore, the importance of studies on such suitable circuit structure is increasing. The design of 4-bit multipliers and our estimation of the performance of larger scale multipliers show that the proposed systolic multiplier achieves comparable latency to an array multiplier with extremely smaller circuit area when the bit-width of input is large.

In Chapter 7, we conclude this dissertation and state future works.

Chapter 2

Preliminaries

2.1 Single-Flux-Quantum (SFQ) Circuits

2.1.1 Josephson Junction

A Josephson junction (JJ) consists of two weakly coupled superconductors which are separated by a very thin insulator or normal conductor as shown in Fig. 2.1. A Josephson junction has the following characteristics called the Josephson effects.

- When a DC current $I < I_C$ (I_C is the critical current of the Josephson junction) is applied to the Josephson junction, no voltage is present on the junction though the current flows. This is the DC Josephson effect.

- When a DC current $I \geq I_C$ is applied, a voltage is present and the junction switches to a finite voltage state. In this case, the phase varies with the voltage that is present

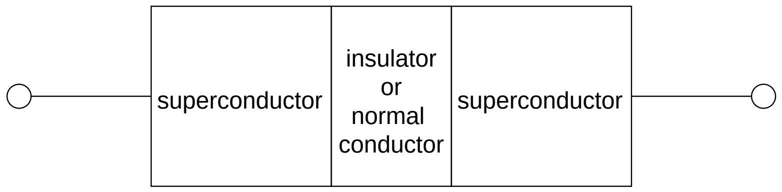


Figure 2.1: A structure of a Josephson junction.

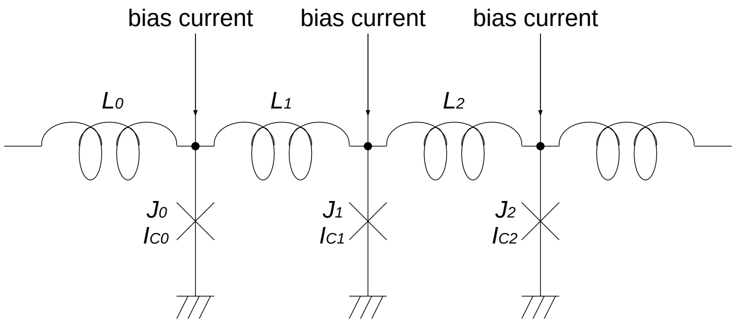


Figure 2.2: A general structure of an SFQ circuit.

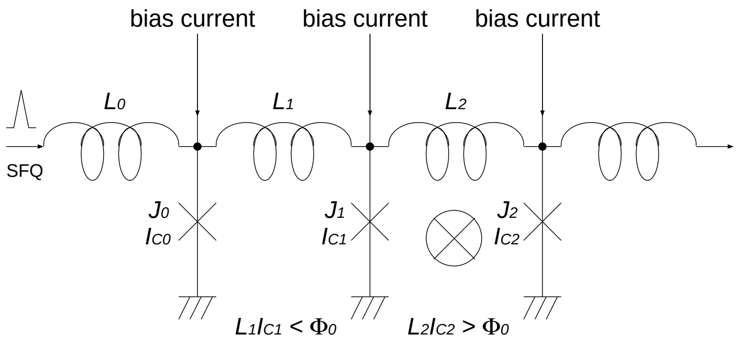


Figure 2.3: A behavior of an SFQ circuit.

on the junction and an AC current whose frequency is proportional to the changes of the phase flows. This is the AC Josephson effect.

2.1.2 SFQ Circuits

An SFQ circuit consists of Josephson junctions and inductances as shown in Fig. 2.2. Here, L indicates an inductance of superconductive loops, J indicates a Josephson junction and I_C indicates the critical current of the Josephson junction. When a Josephson junction of a superconductive loop switches, a flux quantum, i.e., an SFQ, Φ_0 moves to its adjacent loop. In addition, when the junction switches, a very short voltage pulse $V(t)$ called an

SFQ pulse is present. Then,

$$\int V(t)dt = \Phi_0 \simeq 2.07 \quad [\mathrm{mV} \times \mathrm{ps}] \tag{2.1}$$

holds.

In SFQ circuits, an SFQ can be stored in the loop or transmitted to its adjacent loop according to the values of L and I_C. If

$$LI_C < \Phi_0$$

holds, an SFQ is transmitted to its adjacent loop. On the other hand, if

$$LI_C > \Phi_0$$

holds, an SFQ is stored in the loop. When the next SFQ is input, the stored SFQ is transmitted to its adjacent loop. We show a behavior of an SFQ circuit in Fig. 2.3. An SFQ input from the left edge is transmitted from the superconductive loop J_0-L_1-J_1 to J_1-L_2-J_2 because $L_1 I_{C1} < \Phi_0$ holds. In J_1-L_2-J_2, since $L_2 I_{C2} > \Phi_0$ holds, the SFQ is stored in the loop.

2.2 Representation of Logic Values for SFQ Digital Circuits

In SFQ digital circuits, SFQ pulses are used for representing the logic values "1" and "0." When the presence and the absence of a pulse are simply assigned to the logic values "1" and "0," the logic value "0" and the state of "no signal" is indistinguishable. Therefore, in SFQ digital circuits, two methods are used for representing the logic values. One is 'dual-rail representation' and the other is 'synchronous clocking representation.'

2.2.1 Dual-Rail Representation

In dual-rail representation, two signal lines are used for representing two logic values "1" and "0." If there is a pulse on the signal line "1," it represents the logic value "1," while if

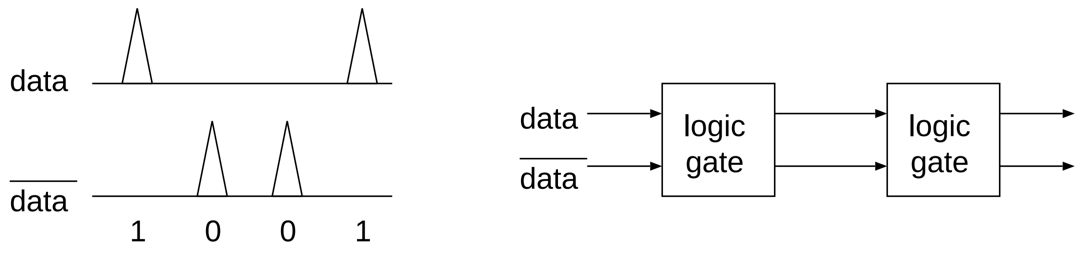


Figure 2.4: A representation of logic values and a general circuit structure using dual-rail representation.

there is a pulse on the signal line “0,” it represents the logic value “0.” Data can be input with the restriction that the data do not overtake previous data in all paths. Therefore, to achieve high operating frequency, arrangement of the delay of all paths is necessary. Process variation, timing jitters and so on may vary the delay. A circuit using dual-rail representation always works correctly by lowering its operating frequency even if such delay variation occurs. However, since two signal lines are necessary for representing the two logic values “1” and “0” of one signal, the circuit area using dual-rail representation tends to be larger. A representation of logic values and a general circuit structure are shown in Fig. 2.4. A circuit using dual-rail representation is called dual-rail circuits.

2.2.2 Synchronous Clocking Representation

In synchronous clocking representation, synchronizing clocks are introduced in order to represent the logic values. The two logic values, i.e., “1” and “0,” are represented by the presence and the absence of a pulse on a data line in a clock period, respectively. A representation of logic values and a general circuit structure are shown in Fig. 2.5. For a gate with clock input (clocked gate), similar to flip-flops (FFs) of CMOS digital circuits, there are constraints between input timing of data and clock signals. One is hold time constraint and the other is setup time constraint. For a clocked gate a, we let $t_{clk}(a)$ be the time of clock input, $t_{data}(a)$ be the time of data input, $\delta_S(a)$ be the setup time, $\delta_H(a)$

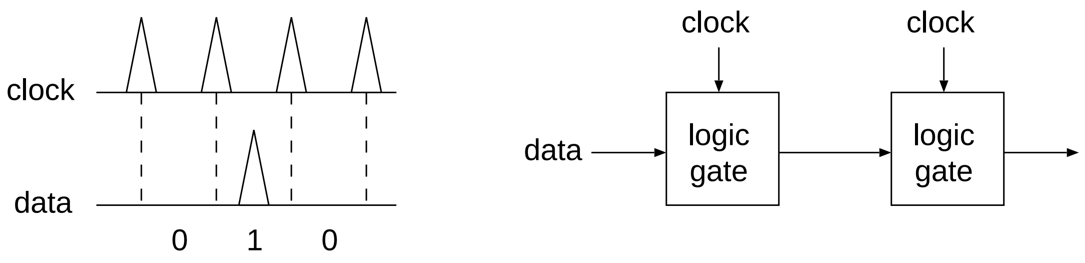


Figure 2.5: A representation of logic values and a general circuit structure using synchronous clocking representation.

be the hold time and T_{CP} be the clock period. Then, the following inequalities have to hold for working the gate correctly.

$$t_{data}(a) \geq t_{clk}(a) + \delta_H(a) \tag{2.2}$$

$$t_{data}(a) \leq t_{clk}(a) + T_{CP} - \delta_S(a) \tag{2.3}$$

Ineq. (2.2) is the hold time constraint and indicates that input of data pulses is prohibited for a certain period, i.e., $\delta_H(a)$, after the input of a clock pulse. Ineq. (2.3) is the setup time constraint and indicates that input of data pulses is prohibited for a certain period, i.e., $\delta_S(a)$, before the input of a clock pulse.

In synchronous clocking representation, there are the following four methods of clock supply[3].

Zero-skew clocking A clock signal is provided for all clocked gates simultaneously. This method is widely used in CMOS digital circuits. A circuit structure with zero-skew clocking is shown in Fig. 2.6. We let $d(i,j)$ be the data delay between gates i and j. Then, the minimum clock period $T^{MIN}_{zero\text{-}skew}$ that is achievable using zero-skew clocking is represented by the following equation:

$$T^{MIN}_{zero\text{-}skew} = \max_{i,j}\{d(i,j) + \delta_S(j)\}.$$

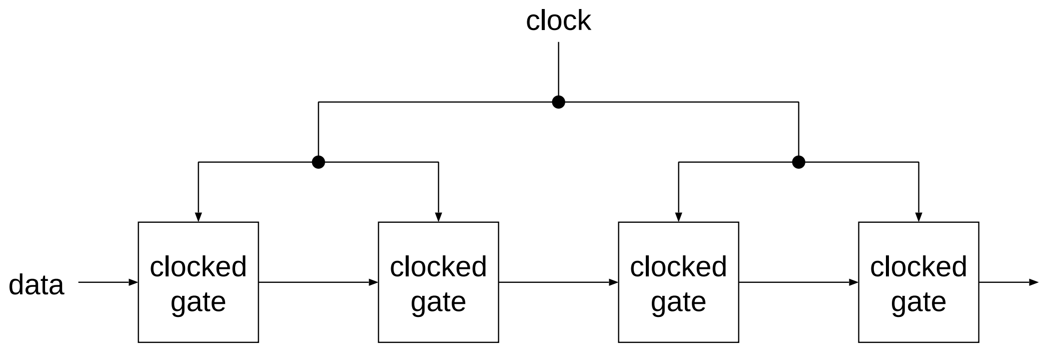


Figure 2.6: A circuit structure with zero-skew clocking.

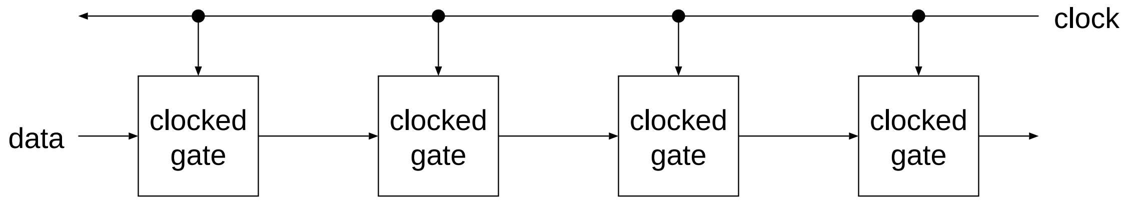


Figure 2.7: A circuit structure with counter-flow clocking.

Counter-flow clocking The direction of clock flow is opposite to that of data flow. A clock signal is input from the output side of the circuit. Since the direction is opposite, no collision occurs between the data of present time and that of former time. In addition, violations on the hold time constraint are not likely to occur in comparison with zero-skew clocking. However, it is difficult to achieve high clock frequency. A circuit structure with counter-flow clocking is shown in Fig. 2.7. We let $c(i,j)$ be the clock delay between gates i and j. Then, the minimum clock period $T^{MIN}_{counter}$ that is achievable using counter-flow clocking is represented by the following equation:

$$T^{MIN}_{counter} = \max_{i,j}\{c(i,j) + d(i,j) + \delta_S(j)\}.$$

Concurrent-flow clocking The direction of clock flow is the same as that of data flow. A clock signal has to arrive at each clocked gate before the arrival of the corresponding data.

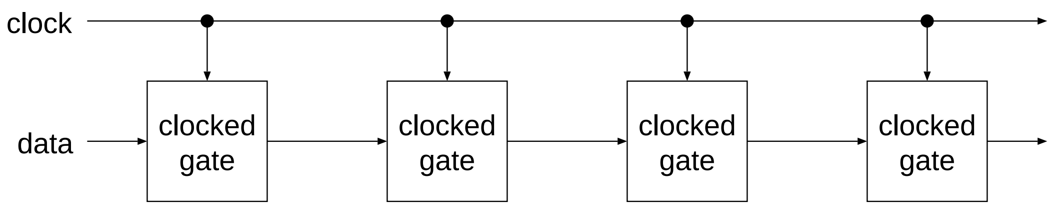


Figure 2.8: A circuit structure with concurrent-flow clocking.

It is possible to make clock frequency high in comparison with zero-skew or counter-flow clocking. However, timing design is difficult. A circuit structure with concurrent-flow clocking is shown in Fig. 2.8. The minimum clock period $T^{MIN}_{concurrent}$ that is achievable using concurrent-flow clocking is represented by the following equation:

$$T^{MIN}_{concurrent} = \max_{j}\{\delta_H(j) + \delta_S(j)\}.$$

Clock-follow-data clocking Similar to concurrent-flow clocking, the direction of clock flow is the same as that of data flow. The different point is that data have to arrive at each clocked gate before the arrival of the corresponding clock signal. One clock pulse carries data from the circuit input to the output. The minimum clock period that is achievable using clock-follow-data clocking is the same as that in concurrent-flow clocking.

Now, concurrent-flow clocking is widely used in SFQ digital circuits because it is possible to make clock frequency high. Malfunction may occur when delays of each path in the circuit vary because of process variation, timing jitters and so on. Therefore, careful timing design is necessary. A circuit using synchronous clocking representation is called synchronous clocking circuits.

Chapter 3

Design Method for Dual-Rail SFQ Digital Circuits

3.1 Introduction

In this chapter, we propose a new method of logic synthesis for dual-rail SFQ digital circuits. Dual-rail circuits have some advantages in comparison with synchronous clocking circuits. For example, timing design of dual-rail SFQ digital circuits is easy in comparison with synchronous clocking SFQ digital circuits. This feature is favorable for developing CAD systems. In addition, since dual-rail circuits always works correctly by lowering those operating frequency even if delay variation occurs, the robustness of circuits is higher. However, the circuit area of dual-rail SFQ digital circuits tends to be larger because two signal lines are necessary for representing the two logic values, i.e., “1” and “0,” and dual-rail representation has not been used. If small-area circuits can be achieved, adoption of dual-rail representation becomes a promising approach for designing SFQ digital circuits. Therefore, it is important to develop logic elements and design methods which can reduce signal lines and circuit area. As a logic element suitable for this aim, a 2×2-Join has been proposed[21, 22]. By using a 2×2-Join, some confluence buffers (CBs)[1] and some splitters (SPLs)[1], we can achieve arbitrary logic operations. An SPL is an element which splits an input pulse to two outputs, while a CB is an elements which merges pulses from two input terminals into one. A 2×2-Join is a universal logic element with a small area for dual-rail SFQ digital circuits.

In the new method to be proposed, we use the 2×2-Join as a logic element. For representation of logic functions, we propose a root-shared binary decision diagram (RSBDD) which is a directed acyclic graph constructed from binary decision diagrams (BDDs)[25, 26]. For reduction of the circuit area, we also propose a new element called resettable 1×2-Join. In the method of logic synthesis, first we construct an RSBDD from given logic functions, and then reduce the number of nodes in the constructed RSBDD by a variable re-ordering technique. Finally, we synthesize a circuit with 2×2-Joins and resettable 1×2-Joins from the reduced RSBDD. For synthesizing larger scale digital circuits, the number of nodes in an RSBDD is an important factor because the number of logic elements included in a synthesized circuit is affected by the number of nodes in the RSBDD. In the variable re-ordering technique, we search for good variable ordering by exchanging nodes in adjacent levels.

For the evaluation of the proposed method, we have implemented the method and have synthesized some benchmark circuits. The experimental results show that the proposed method can synthesize dual-rail SFQ digital circuits which consist of about 27.1% fewer logic elements than those synthesized by a Transduction-based method.

This chapter is organized as follows. In Section 3.2, we describe dual-rail SFQ circuits with 2×2-Joins and propose a new logic element, i.e., resettable 1×2-Join, for reduction of the circuit area. In Section 3.3, we propose a root-shared binary decision diagrams (RSBDDs) and in Section 3.4, we propose a method of logic synthesis. In Section 3.5, we show experimental results. Finally, in Section 3.6, we give the summary of this chapter.

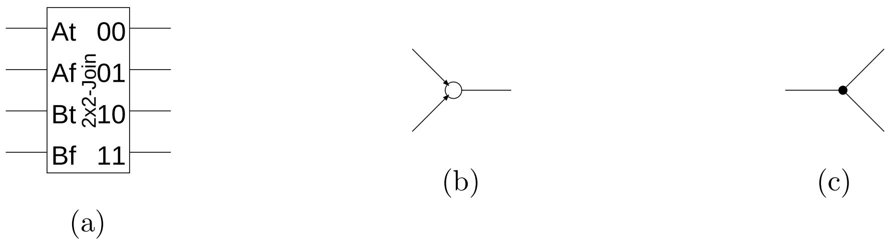


Figure 3.1: Notations of basic elements: (a) 2×2-Join, (b) CB, (c) SPL.

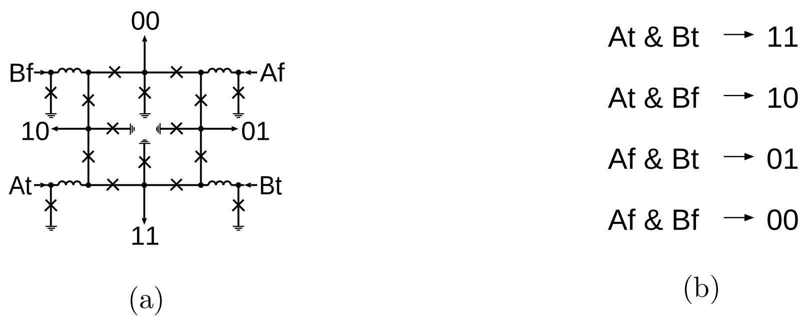


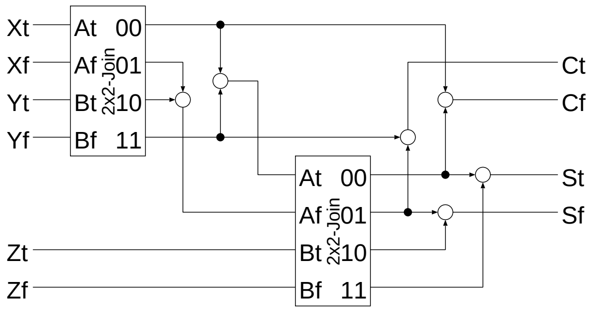
Figure 3.2: 2×2-Join: (a) JJ schematic, (b) input-output relations.

3.2 Dual-Rail SFQ Circuits with 2×2-Joins

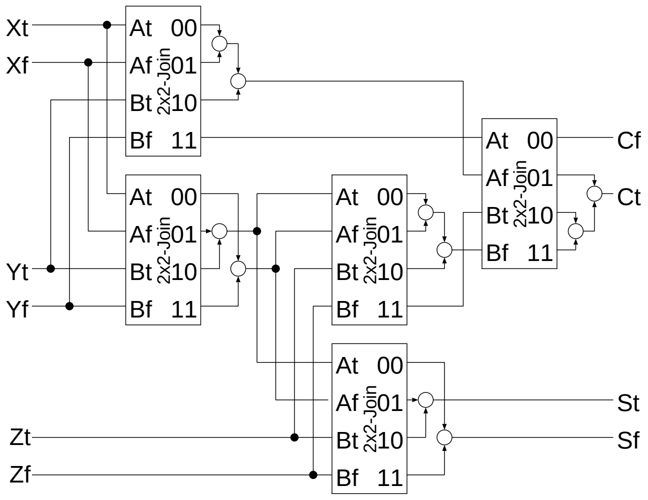
3.2.1 Dual-Rail SFQ Circuits with 2×2-Joins

In dual-rail representation, two signal lines are used for representing a logic signal. If there is a pulse on the signal line "0" ("*false*-line"), it represents the logic value "0," while if there is a pulse on the signal line "1" ("*true*-line"), it represents the logic value "1." We can input data with the restriction that the data do not overtake previous data in all paths. Therefore, we can achieve high operating frequency by arranging the delay time of all paths.

A 2×2-Join[21, 22] is a logic element for dual-rail SFQ digital circuits. By using a 2×2-Join, some confluence buffers (CBs)[1] and some splitters (SPLs)[1], we can achieve arbitrary logic operations. An SPL is an element which splits an input pulse to two outputs, while a CB is an elements which merges pulses from two input terminals into one. Two input pulses for a CB are not allowed to arrive at the same time. In Fig. 3.1,



(a)



(b)

Figure 3.3: Two design examples of a full adder using 2×2-Joins: (a) the circuit with the fewest number of 2×2-Joins, (b) the circuit consists of AND, OR and XOR using 2×2-Joins and CBs.

we show notations of the basic elements that we use in this chapter.

In Fig. 3.2, we show a JJ schematic and input-output relations of a 2×2-Join. Each cross in Fig. 3.2 shows a JJ. Dual-rail data are fed into inputs A and B of a 2×2-Join, and according to the combination of the input values, a pulse is generated at one of the 4 outputs (00, 01, 10, 11). For example, if pulses are fed into At and Bt, a pulse is generated at output 11. A 2×2-Join is a universal logic element with a small area.

As an example of dual-rail SFQ circuits with 2×2-Joins, we show full adders ($S =$

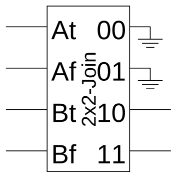
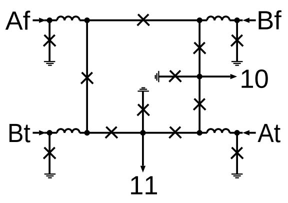


Figure 3.4: A 2×2-Join where two of 4 outputs are not used.



(a)

At & Bt $\rightarrow$ 11

At & Bf $\rightarrow$ 10

Af & Bt $\rightarrow$ (reset)

Af & Bf $\rightarrow$ (reset)

(b)

Figure 3.5: Resettable 1×2-Join: (a) JJ schematic, (b) input-output relations.

$X \oplus Y \oplus Z$, $C = X \cdot Y + Y \cdot Z + Z \cdot X$) in Fig. 3.3. Figure 3.3 (a) shows the circuit with the fewest number of 2×2-Joins. On the other hand, Fig. 3.3 (b) shows the circuit that consists of AND, OR and XOR using 2×2-Joins and CBs. The smallest circuit shown in Fig. 3.3 (a) consists of only two 2×2-Joins, while the circuit shown in Fig. 3.3 (b) consists of 5. By using 2×2-Joins effectively, we can design small-area dual-rail circuits.

3.2.2 Resettable 1×2-Join

In a circuit synthesized with 2×2-Joins, CBs and SPLs, there are 2×2-Joins whose two of 4 outputs are not used as is shown in Fig. 3.4. For reduction of the circuit area, we propose a new logic element called resettable 1×2-Join by removing 2 outputs from a 2×2-Join.

A resettable 1×2-Join is a logic element with 4 inputs and 2 outputs. We show a JJ schematic and input-output relations, and a notation of a resettable 1×2-Join in Fig. 3.5 and Fig. 3.6, respectively. A resettable 1×2-Join consists of only 12 JJs in comparison with 16 JJs of a 2×2-Join. When pulses are fed into At and Bt, a pulse is generated

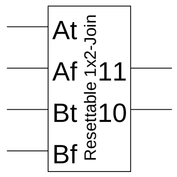


Figure 3.6: Notation of a resettable 1×2-Join.

at output 11 and when pulses are fed into At and Bf, a pulse is generated at output 10. Different from a 2×2-Join, when pulses are fed into Af and Bf/Bt of a resettable 1×2-Join, no pulse is generated and the resettable 1×2-Join is reset.

We have confirmed that a 2×2-Join and a resettable 1×2-Join have almost the same physical characteristics (delay time, operation margins and so on) by an analog circuit simulation.

3.3 Root-Shared Binary Decision Diagram

3.3.1 Binary Decision Diagram (BDD)

A binary decision diagram (BDD) on a variable set $X = \{x_1, \ldots, x_v\}$ is a directed acyclic graph and represents a logic function $f : \{0,1\}^v \to \{0,1\}$. As nodes, there are non-terminal nodes labeled by variable x_i and terminal nodes labeled by the constant "0" or "1." The non-terminal nodes are called variable nodes and the terminal nodes are called leaves. Among variable nodes, there is a root node whose indegree is zero. The outdegree of a variable node is two, while that of a leaf is zero. Each of the two outgoing edges of a variable node has a label. An edge labeled by "0" is called 0-edge and one labeled by "1" is called 1-edge. For an input values of the logic function $a = (a_1, \ldots, a_v) \in \{0,1\}^v$, we can trace the BDD from the root node in accordance with the values of the input. From a node labeled by x_i, we trace the 0-edge of the node when $a_i = 0$, while we trace the 1-edge when $a_i = 1$. The value of the leaf that we reach is the value of the logic function with the input values. The BDD can be constructed by applying Shannon expansion to

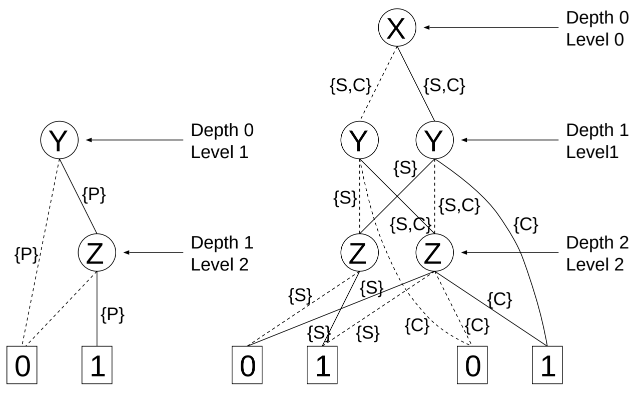


Figure 3.7: An example of an RSBDD.

the logic function.

We can represent a logic function uniquely using a BDD when variable ordering is fixed, redundant nodes are removed and equivalent nodes are merged. A redundant node is a node whose 0-edge and 1-edge point to the same node. Two nodes are equivalent when they have the same label, the 0-edges of them point to the same node and the 1-edges of them point to the same node. Such BDD is called reduced ordered BDD (ROBDD)[26]. BDDs are widely used in electronic design automation (EDA) fields.

3.3.2 Root-Shared Binary Decision Diagram (RSBDD)

An RSBDD on a variable set $X = \{x_1, \dots, x_v\}$ is a directed acyclic graph and represents a set of logic functions $F = \{f_1, \dots, f_m\}$ ($f_i : \{0,1\}^{v_i} \to \{0,1\}$, $v_i \leq v$). As nodes, there are variable nodes and leaves. Among variable nodes, there are root nodes. Each edge has two labels F' ($F' \subseteq F$) and c ($c \in \{0,1\}$). We call the former label ‘f-label’ and the latter ‘c-label.’ For a function f_i and an input values $a = (a_1, \dots, a_v) \in \{0,1\}^v$, we can trace the RSBDD from one of the root nodes. We trace the edges whose f-labels contain f_i. From a node labeled by x_i, we trace the 0-edge of the node when $a_i = 0$, while we trace the 1-edge when $a_i = 1$. The value of the leaf that we reach is the value of f_i with the input values.

We show an example of an RSBDD in Fig. 3.7. The RSBDD represents functions $S(X, Y, Z) = X \oplus Y \oplus Z$, $C(X, Y, Z) = X \cdot Y + Y \cdot Z + Z \cdot X$ and $P(Y, Z) = Y \cdot Z$. In Fig. 3.7, dashed edges show that c-labels of the edges are 0 and solid edges show that c-labels are 1. "$\{S, C\}$," "$\{S\}$," "$\{C\}$" and "$\{P\}$" are f-labels. The RSBDD consists of two connected parts. The left side part represents $\{P\}$ and the right side part represents $\{S, C\}$.

An RSBDD is constructed by the following rule from BDDs. Variable ordering of all the BDDs is identical. For given variable ordering, if labels of root nodes of the BDDs are not unique, a disconnected graph is constructed.

[The rule for constructing an RSBDD of a set of functions F]

- If the size of F is one, the RSBDD of F is obtained by adding f-labels to the BDD representing the function.
- For $F = F_1 \cup F_2$, the RSBDD of F is obtained by merging nodes and edges of the RSBDDs of F_1 and F_2 that satisfy the followings.
    - Node u_1 of the RSBDD of F_1 and node u_2 of the RSBDD of F_2 are mergeable when their labels are identical and either of the following conditions holds.
        1. They are root nodes, or
        2. There is one-to-one relation between the sets of parent nodes of nodes u_1 and u_2 and, all the corresponding parent nodes are mergeable and the c-labels of the outgoing edges from the related parent nodes to nodes u_1 and u_2 are identical.
    - Edges e_1 and e_2 are mergeable when their start nodes are identical, their end nodes are identical and their c-labels are identical. e_1 and e_2 are to be merged by taking the union of their f-labels.

We define the depth of a root node as 0 and the depth of a variable node except root nodes as the largest depth of its parent nodes plus 1 and the level of a node as the position

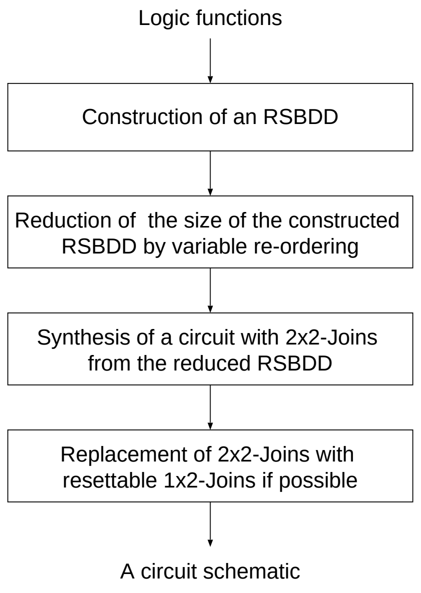

Logic functions
Construction of an RSBDD
Reduction of the size of the constructed RSBDD by variable re-ordering
Synthesis of a circuit with 2x2-Joins from the reduced RSBDD
Replacement of 2x2-Joins with resettable 1x2-Joins if possible
A circuit schematic

Figure 3.8: Flow of logic synthesis.

of the variable of the node in the variable ordering. In the right side part of Fig. 3.7, the depths of the nodes labeled by X, Y and Z are 0, 1 and 2, respectively and in the left side part, the depths of the nodes labeled by Y and Z are 0 and 1, respectively. The levels of the nodes labeled by X, Y and Z are 0, 1 and 2, respectively. We define the size of an RSBDD as the total number of variable nodes and also define the width of a level as the number of nodes in the level. In Fig. 3.7, the size is 7 and the width of level 0, 1 and 2 are 1, 3 and 3, respectively.

3.4 Method of Logic Synthesis

3.4.1 Flow of Logic Synthesis

Now, we propose a method of logic synthesis. We show the flow of the proposed method in Fig. 3.8. We first construct an RSBDD from given logic functions, and then reduce the size of the constructed RSBDD by a variable re-ordering technique. We synthesize a circuit that uses only 2×2-Joins, CBs and SPLs from the reduced RSBDD. Finally, we replace 2×2-Joins with resettable 1×2-Joins if possible for reduction of the circuit area.

Variable ordering of an RSBDD is important because the number of nodes in the RSBDD is affected by the variable ordering of it. Exact minimization of an RSBDD is almost impossible because the computation cost of exact minimization is very high. Therefore, we apply the variable re-ordering technique in [27] to an RSBDD to reduce its size.

3.4.2 Construction of an RSBDD

Method of RSBDD Construction

When functions are given by logic expressions, we can construct an RSBDD according to the given functions incrementally.

First, we select one logic function and construct the RSBDD representing the function. Then, we select unprocessed functions one by one and construct the RSBDD from the selected function. If there is a node which is newly created or one whose number of parent nodes varies or one whose parent nodes are merged with other nodes, we check whether mergeable pairs exist and merge all mergeable pairs.

Example of RSBDD Construction

As an example of RSBDD construction, we show the process of constructing an RSBDD of the full adder function $S(X, Y, Z) = X \oplus Y \oplus Z$ and $C(X, Y, Z) = X \cdot Y + Y \cdot Z + Z \cdot X$ in Fig. 3.9. We assume the variable ordering is X-Y-Z. In Fig. 3.9, for explanation, we numbered the nodes. We process the functions sequentially.

First, we construct the BDD of function S and label all edges of the BDD "$\{S\}$" as f-labels. Figure 3.9 (a) is the RSBDD of $\{S\}$. Then, we process function $C = X \cdot Y + Y \cdot Z + Z \cdot X$. Figure 3.9 (b) shows an intermediate step of processing X. Here, nodes 0 and 5 are root nodes and labels of these two nodes are identical. Therefore, we can merge these two nodes. By merging them, we can obtain the RSBDD shown in Fig. 3.9 (c). We

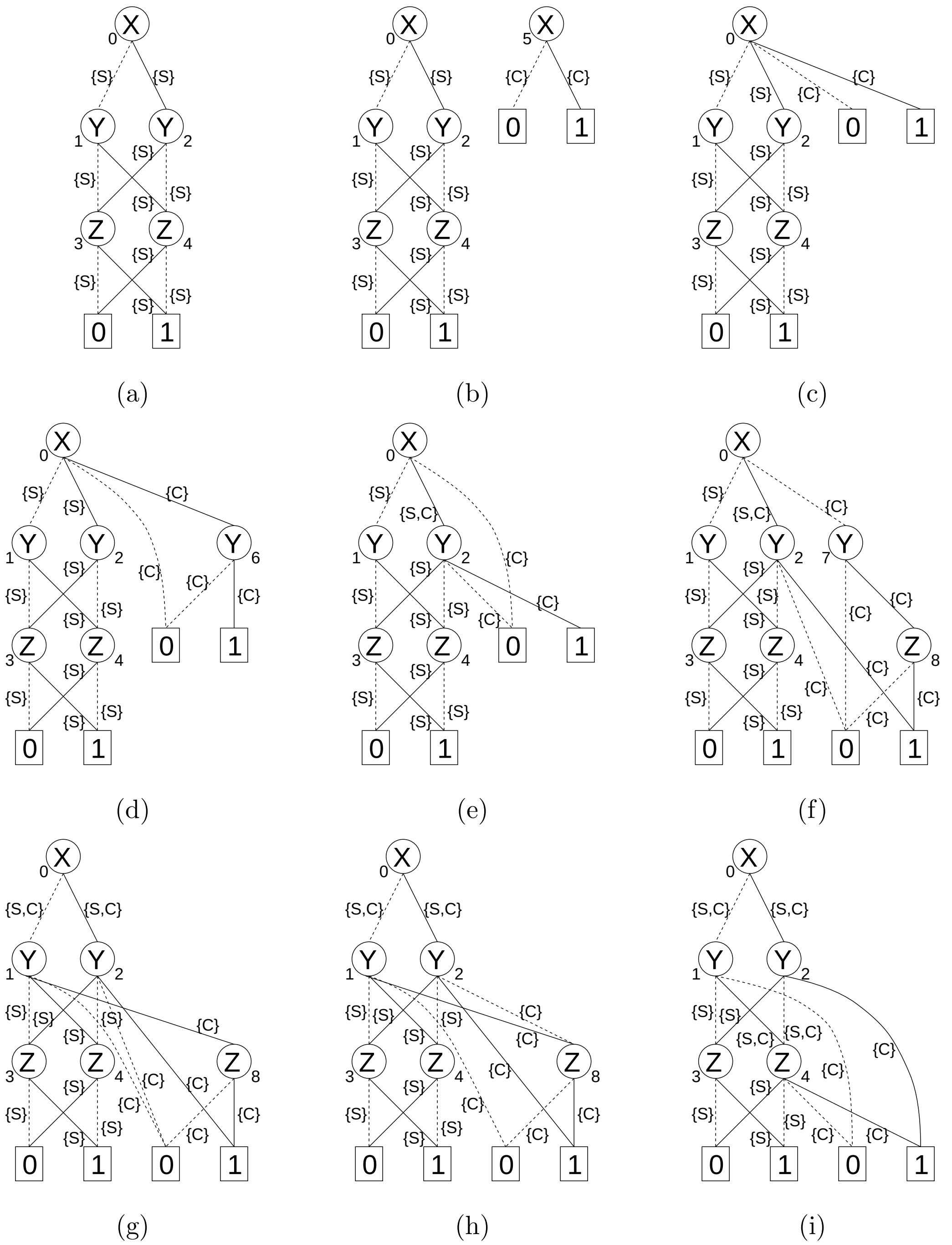


Figure 3.9: Process of constructing RSBDD of the full adder function.

have shown Fig. 3.9 (b) for explanation, although the RSBDD shown in Fig. 3.9 (b) is not constructed actually and the RSBDD shown in Fig. 3.9 (c) is directly constructed from

Fig. 3.9 (a).

Figure 3.9 (d) shows an intermediate step of processing $X \cdot Y$. Nodes 2 and 6 have the same label. Furthermore, c-labels of the edges between these two nodes and node 0, that is their parent node, are both 1. Therefore, nodes 2 and 6 are mergeable. By merging them, we can obtain the RSBDD shown in Fig. 3.9 (e).

Figure 3.9 (f) shows an intermediate step of processing $X \cdot Y + Y \cdot Z$. In this step, we can merge nodes 1 and 7, and can obtain the RSBDD shown in Fig. 3.9 (g).

Figure 3.9 (h) shows an intermediate step of processing $X \cdot Y + Y \cdot Z + Z \cdot X$. In this step, nodes 4 and 8 are mergeable. Their labels are the same, i.e., Z, and c-labels of the edges between these two nodes and node 1 which is one of the parent nodes are the same, i.e., 1, and the edges between these two nodes and node 2 are the same, i.e., 0. By merging these two nodes, we can obtain the RSBDD shown in Fig. 3.9 (i). Similar to Fig. 3.9 (b), the RSBDDs shown in Fig. 3.9 (d), (f) and (h) are not constructed, but the RSBDDs shown in Fig. 3.9 (e), (g) and (i) are directly constructed.

Computational Complexity of RSBDD Construction

Now, we consider the computational complexity of constructing an RSBDD. When we check whether two nodes are mergeable, we have to check all parent nodes of the two nodes. The number of parent nodes of a node is the indegree of it. The summation of the number of parent nodes of the nodes in an RSBDD is the summation of their indegrees, which is the number of edges in the RSBDD. For a connected part of the RSBDD, a node is checked at most for all nodes in the same level of the part. Therefore, the computational complexity of constructing an RSBDD is $O(f \cdot e \cdot w_{\max})$ where f is the number of given logic functions, e is the number of edges in the RSBDD, and $w_{\max}$ is the maximum width of the connected part of the RSBDD. The computational complexity is $O(fe^2)$.

3.4.3 Reduction of the Size of an RSBDD

Method of Size Reduction

By searching for good variable ordering and changing variable ordering, we can reduce the size of an RSBDD. The number of root nodes in an RSBDD may vary when variable ordering of it changes. In the proposed method of logic synthesis, we apply the variable re-ordering technique in [27], that is originally proposed for conventional BDDs, to the RSBDD constructed in Section 3.4.2 and reduce the size. We exchange all nodes in adjacent levels by level exchange. We determine strategies of level exchange and reduce the size of the RSBDD.

When nodes in adjacent levels are exchanged, all the nodes have to be shared by the same set of functions. If the nodes are shared by a different set of functions, we cancel the sharing. Furthermore, if we exchange nodes in levels i and $i+1$, we have to split nodes in level $i+1$ whose parent nodes are in levels lower than i. After these processes are carried out, the nodes in level i and those in level $i+1$ can be exchanged. After exchange of the nodes, we check whether mergeable pairs exist and if such pairs exist, merge them pairwise.

Example of Size Reduction

As an example, we show the level exchange of an RSBDD in Fig. 3.10. Figure 3.10 (a) shows the original RSBDD. We exchange the nodes in level i and those in level $i+1$, that is, the nodes labeled by Y and those labeled by Z. First, we cancel the sharing of functions F and G because the right side node in level $i+1$ is not shared by F and G. Furthermore, since a parent node of the right side node in level $i+1$ is in level $i-1$, we split the right side node. We show the RSBDD after the cancellation and the splitting in Fig. 3.10 (b). Then, we exchange the nodes in level i and those in level $i+1$. Figure 3.10 (c) shows the RSBDD after the exchange. Finally, we check whether mergeable pairs exist and merge all mergeable pairs. Figure 3.10 (d) shows the RSBDD after the merging.

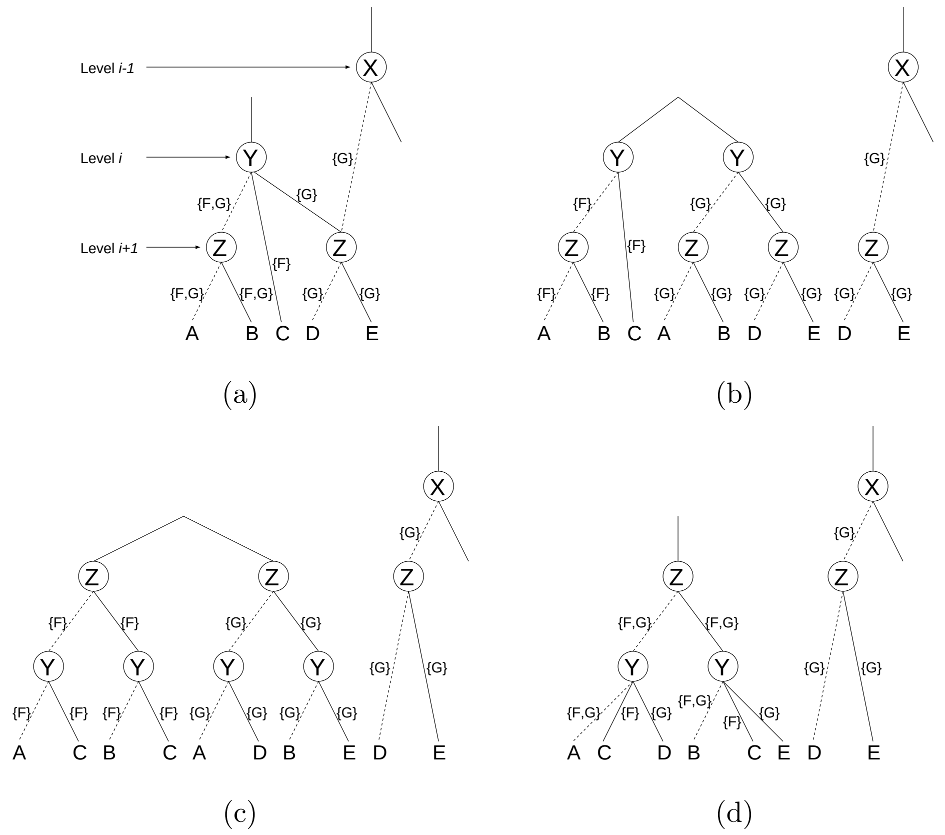


Figure 3.10: Example of level exchange: (a) an original RSBDD, (b) after splitting a node and cancellation of sharing functions, (c) after exchange of level i and level $i+1$, (d) after merging of mergeable pairs.

As a strategy of level exchange, we adopt sifting[27]. We briefly describe sifting here. We assume that variables of a BDD are $(x_1,\ldots,x_i,\ldots,x_v)$. When we fix variable ordering except x_i, we can insert x_i into v different positions. We search for the position that the size of the BDD is the minimum and move the variable to the position. We select all variables one by one and reduce the size.

Computational Complexity of Size Reduction

Here, we consider the computational complexity of the level exchange for a connected part of an RSBDD and sifting.

First, we consider the computational complexity of the level exchange of level i and level $i+1$. We assume the width of level i in an original RSBDD to be w_i, the width of level $i+1$ to be w_{i+1} and the number of variables to be v. The computational complexity of splitting nodes is $O(p_{i+1})$ where p_{i+1} is the number of parent nodes of nodes in level $i+1$, and the computational complexity of exchanging nodes is $O(f \cdot (w_i + w_{i+1}))$. The computational complexity of the cancellation of sharing functions is $O(f \sum_{x=i}^{v} n_x)$ where n_i is the number of nodes in level i. The computational complexity of the merging is $O(f \cdot w_{\max} \sum_{x=i}^{v} e_x)$ where e_i is the number of edges in level i. Therefore, the computational complexity of level exchange of level i and $i+1$ is $O(f \cdot w_{\max} \sum_{x=i}^{v} e_x)$. The computational complexity is $O(fe^2)$.

In sifting, selection of variables are carried out v times. The number of level exchange for each selection is $3(v-1)$ in the worst case. Therefore, in the worst case, the number of level exchange is $3v(v-1)$.

3.4.4 Synthesis of a Circuit

Method of Circuit Synthesis

We synthesize a circuit from the RSBDD constructed up to Section 3.4.3. We select connected parts from the RSBDD one by one and synthesize the circuit. Here, we describe a method for synthesizing a circuit block from a connected part of the RSBDD. We describe an example of circuit synthesis later using Fig. 3.11.

We process sequentially from the root node to the leaves. We construct the first stage of the block from the root node and the nodes of depth 1 using 2×2-Joins. Then, we construct the i-th ($i \geq 2$) stage from the nodes of depth i. Finally, we derive circuit outputs from the leaves.

To select an appropriate signal from the 4 outputs of a 2×2-Join, we label a number '0' or '1' to nodes of the RSBDD. We call the number 'output-selection-number.' When the output-selection-number is 0, we select output 00 or 01 of the corresponding 2×2-Join

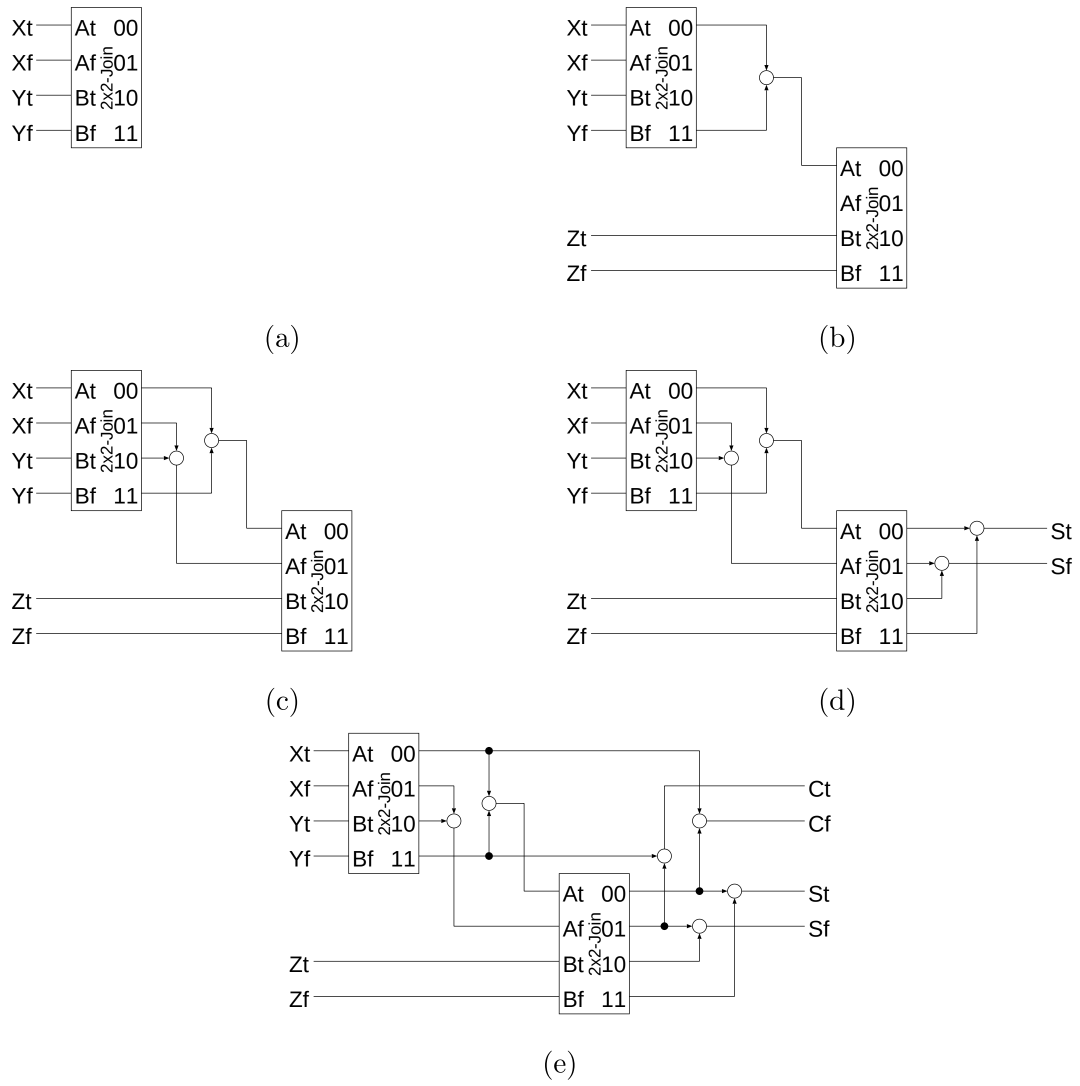


Figure 3.11: Process of synthesizing a full adder.

according to the c-label of the edge of the corresponding node. For example if the c-label of the edge is 0, we select 00. Similarly, when the output-selection-number is 1, we select output 10 or 11.

We construct the first stage from the root node and the nodes of depth 1. For nodes of depth 1 that have the same label, we place a 2×2-Join. We connect the *true*-line of the root variable to input At of the 2×2-Join, the *false*-line of the root variable to Af, the *true*-line of the variable of depth 1 to Bt and the *false*-line of the variable of depth 1 to

Bf. We mark the root node and the processed nodes of depth 1 with “processed.” We let the output-selection-number of the node of depth 1 be 0 or 1 according to the edge of the root node that points to the node being 0 or 1, respectively.

We construct the i-th ($i \geq 2$) stage from the nodes of depth i. For a node of depth i, we place a 2×2-Join and connect the *true*-line and the *false*-line of the variable of depth i to input Bt and Bf of the 2×2-Join, respectively. We connect intermediate circuit outputs that are constructed up to the previous stages to input At and Af as follows.

To input At, we connect lines that correspond to edges pointing to the processing node. We merge these lines to one line using CBs and connect it to At. When a parent node of the processing node is the root node, we select the *false*-line or the *true*-line of the root variable according to the edge between the parent node and the processing node being a 0-edge or a 1-edge, respectively. When the parent node of the processing node is not the root node, we select an output of the 2×2-Join that corresponds to the parent node according to the output-selection-number and the c-label of the edge. From the 4 outputs (00, 01, 10, 11), we select an output whose label is identical to the concatenation of the output-selection-number and the c-label of the edge.

To input Af, we connect lines corresponding to edges that are outgoing from nodes included in paths from the root node to the processing node and are not included in the paths. Selection of lines is carried out in the same way as the case of At. We check whether there are edges that point to an unprocessed node of the same level and that are identical to the edges selected for connecting to Af. If such edges exist, we set the output-selection-number of the unprocessed node to 0 and mark the unprocessed node with “processed.” When such an unprocessed node exists, a 2×2-Join is shared by two nodes.

We set the output-selection-number of the processing node to 1 and mark the node with “processed.” We apply these processes to all unprocessed nodes.

We derive circuit outputs from the leaves of the connected part of the RSBDD. The *false*-lines are derived from 0-leaves and the *true*-lines are derived from 1-leaves. We select lines in the same way as the case of At.

The number of stages of 2×2-Joins in the synthesized circuit block is the maximum depth of the connected part of the RSBDD and the number of 2×2-Joins is less than or equal to the number of nodes except the root node. In the best case, the number of 2×2-Joins is half of the number of nodes in the connected part of the RSBDD.

Example of Circuit Synthesis

As an example of circuit synthesis, we consider a full adder. The RSBDD of a full adder is shown in Fig. 3.9 (i). It consists of one connected part. We show process of the synthesis in Fig. 3.11.

First, we construct the 2×2-Join of the first stage from the root node and the two nodes of depth 1. We prepare one 2×2-Join and connect the *true*-line *(Xt)* and the *false*-line *(Xf)* of the root variable to input At and Af, respectively and, the *true*-line *(Yt)* and the *false*-line *(Yf)* of the variable of depth 1 to Bt and Bf, respectively. We mark the root node and the two nodes of depth 1 with "processed." We set the output-selection-number of node 1 and node 2 to "0" and "1," respectively. We show the circuit that is synthesized up to this step in Fig. 3.11 (a).

Then, we construct the 2×2-Joins of the second stage from nodes of depth 2. Here, we process left side node (node 3) first. We prepare a new 2×2-Join and connect the *true*-line *(Zt)* and the *false*-line *(Zf)* of the variable of depth 2 to input Bt and Bf, respectively. We mark the node with "processed" and set the output-selection-number to "1."

To input At, we connect lines that correspond to edges pointing to the node (0-edge of node 1 and 1-edge of node 2). Since the output-selection-number of node 1 is 0 and the output-selection-number of node 2 is 1, output 00 and 11 of the 2×2-Join of the first stage are the corresponding lines. We merge these two lines to one line using a CB and connect the merged line to At. The circuit shown in Fig. 3.11 (b) is synthesized up to this step.

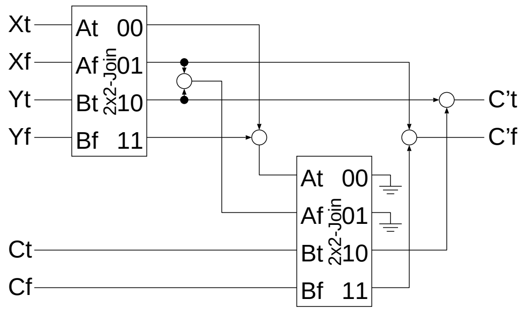
To input Af, we connect lines corresponding to edges that are outgoing from nodes included in the paths from the root node to the processing node and are not included in the paths (1-edge of node 1 and 0-edge of node 2). Therefore, we connect output 01 and 10 of the 2×2-Join of the first stage to Af. These lines are identical to lines corresponding to edges pointing to node 4. We set the output-selection-number of node 4 to "0" and mark node 4 with "processed." We complete processing of the nodes of depth 2. The circuit shown in Fig. 3.11 (c) is synthesized up to this step. In this example, we can construct the second stage by using only one 2×2-Join.

We derive circuit outputs from the leaves. By constructing circuit outputs of function *S (St, Sf)*, we can synthesize the circuit shown in Fig. 3.11 (d). We also derive circuit outputs of function *C (Cf, Ct)* and can synthesize a full adder shown in Fig. 3.11 (e).

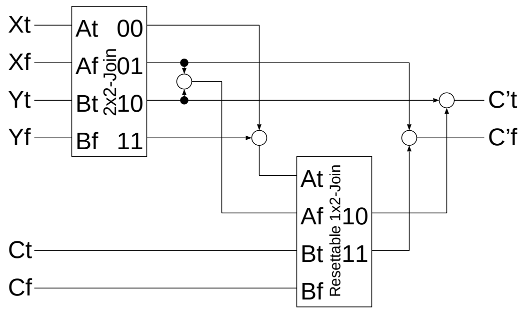
The number of 2×2-Joins in the full adder is 2 and the number of stages of 2×2-Joins is 2 because the maximum depth of the RSBDD is 2.

Computational Complexity of Circuit Synthesis

First, we consider the computational complexity of synthesizing a circuit block from a connected part of an RSBDD. When we search for signal lines that are connected to Af, we have to check all ancestor nodes of the processing node. However, by preserving all intermediate results, that is, edges that are included in the paths from the root node to the processing node and edges that are not in the paths, we can search for the signal lines by only checking the parent nodes of the processing node. The number of edges that are preserved in the processing node is less than or equal to the number of all edges in the connected part. Therefore, the computational complexity of searching for the signal lines that are connected to Af is $O(e_{\text{connected-part}} \cdot q_a)$ where $e_{\text{connected-part}}$ is the number of edges in the connected part and q_a is the number of parent nodes of the processing node. The summation of the number of all parent nodes that are included in a connected part is identical to the number of edges in the connected part. Therefore, the computational complexity of searching for all signals is $O(e^2_{\text{connected-part}})$, and this is the computational complexity of synthesizing the circuit block.



(a)



(b)

Figure 3.12: Example of replacement with a resettable 1×2-Join: (a) before replacement, (b) after replacement with a resettable 1×2-Joins.

When we synthesize a circuit from an RSBDD, the computational complexity is proportional to the summation of the computational complexity of synthesizing each circuit block, i.e., $O(\sum_{c=1}^{g} e^2_{\text{connected-part } c})$ where g is the number of connected parts of the RSBDD. The computational complexity is $O(e^2)$.

3.4.5 Replacement of 2×2-Joins with Resettable 1×2-Joins

In a circuit synthesized up to Section 3.4.4, there can be 2×2-Joins where two of the 4 outputs are not used, as shown in Fig. 3.4. By replacing such 2×2-Joins with resettable 1×2-Joins, we can reduce the circuit area. Since a resettable 1×2-Join has almost the same physical characteristics as a 2×2-Join, we can reduce the circuit area by simply replacing the 2×2-Joins shown in Fig. 3.4 with resettable 1×2-Joins. We show an example of the replacement in Fig. 3.12.

3.5 Experimental Results

3.5.1 Evaluation of Variable Re-ordering techniques

For evaluating the employed method of variable re-ordering, we have implemented the method by programming language C. In the program, we construct an RSBDD from given logic functions and reduce the size by level exchange with sifting. We show the experimental results on some benchmark circuits of LGSynth'91[28] in Table 3.1. For comparison of the method, we have also implemented other strategies "random" and "window permutation[27]." In the random and window permutation, we have selected a level at random 1,000 times. In addition, we set window size as 3 in the window permutation. The experimental environment is a SunBlade2000 with UltraSPARC-III+ 1.2 GHz CPU and 2 GByte memory.

The experimental results show that sifting is the best among the three strategies. Execution time of the sifting is short in comparison with other strategies and in many cases, the size of the RSBDDs with sifting is the best.

3.5.2 Evaluation of the Proposed Method of Logic Synthesis

For evaluating the proposed method of logic synthesis, we have implemented the method by programming language C. We have synthesized some benchmark circuits in LGSynth'91 benchmark set[28]. In the program, we construct an RSBDD from given logic functions first. Then, we reduce the size of the constructed RSBDD by level exchange with sifting and synthesize a circuit using the reduced RSBDD. In the program, since the number of logic elements does not vary, we have used only 2×2-Joins as logic elements.

In [20], a BDD-based design method was proposed as another design method of dual-rail SFQ circuits. In the design method, a binary switch called Bina[20] is used as a logic

Table 3.1: Experimental results of variable re-ordering.

circuit name	# of inputs	# of outputs	initial size	sifting		random		window permutation	
				size	CPU time (s)	size	CPU time (s)	size	CPU time (s)
alu2	10	6	200	150	9.9	139	40.1	158	160.5
alu4	14	8	1363	355	105.9	1151	449.9	535	475.7
cmb	16	4	35	27	6.9	27	18.4	35	110.4
f51m	8	8	35	35	3.9	34	31.4	34	97.8
lal	26	19	77	57	24.2	57	24.9	59	127.4
term1	34	10	734	82	74.2	200	82.1	83	187.2
ttt2	24	21	179	106	25.0	139	31.2	140	126.3
x3	135	99	2154	497	1185.2	1942	547.0	1677	2532.9
z4ml	7	4	13	13	2.4	13	21.7	13	154.1

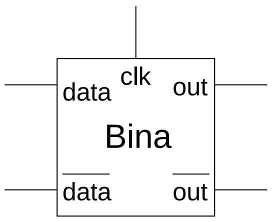


Figure 3.13: Bina.

element. We show a notation of Bina in Fig. 3.13. If a clock pulse is input to *clk* after input of a pulse to *data*, a pulse is generated at *out*. Similarly, if a clock pulse is input to *clk* after input of a pulse to $\overline{data}$, a pulse is generated at $\overline{out}$. A set of BDDs is constructed by given logic functions and the set of BDDs are merged. Then, each node of the merged BDDs is replaced by Bina one by one. As an example, we show a full adder with Binas[20] in Fig. 3.14. There are 4 Binas in the circuit. On the other hand, we can construct a full adder with only 2 2x2-Joins (Fig. 3.3). Furthermore, a 2x2-Join cell included in the SFQ cell library[5] consists of 20 JJs in comparison with 27 JJs of a Binas. A 2x2-Join is a

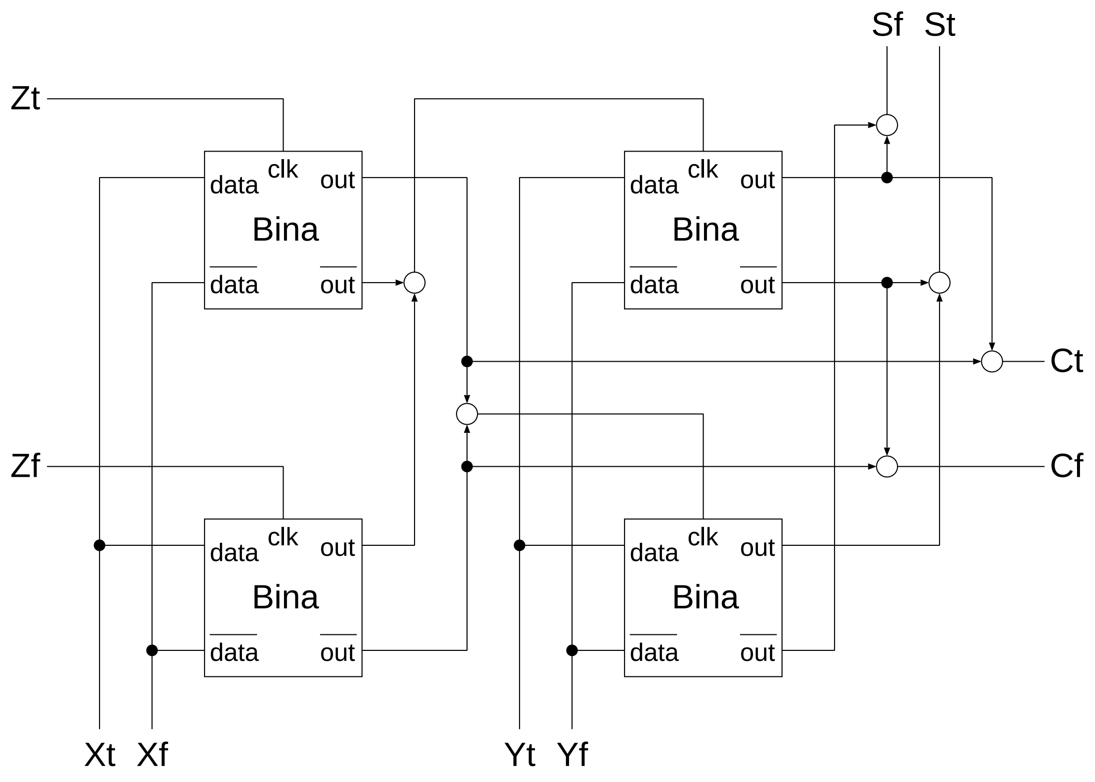


Figure 3.14: A full adder using Binas.

smaller logic element than a Bina.

In [24], a Transduction-based method was proposed as another method of logic synthesis for dual-rail SFQ circuits with 2×2-Joins. In the design method, a new logic element called 2x2-AND/XOR[24] shown in Fig. 3.15 is constructed from a 2x2-Join and initial circuits are constructed by the 2x2-AND/XOR. Then, the circuits are optimized by using a transformation-based heuristic method based on the Transduction Method.

We have compared the proposed method with the above methods. In [20], a merging method of BDDs is not described. We have used the constructing method of RSBDDs for merging BDDs. We show the experimental results in Table 3.2. The numbers of 2×2-Joins of the method in [24] is the values described in [24].

From the experimental results, our method can synthesize circuits with fewer 2×2-

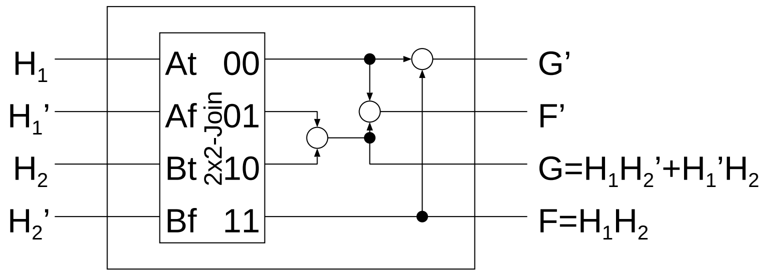


Figure 3.15: 2x2-AND/XOR.

Table 3.2: Experimental results of logic synthesis.

circuit name	# of inputs	# of outputs	our method				method in [24]	ratio (%) (our / in [24])	method in [20]
			# of 2×2-Joins	# of CBs	# of SPLs	CPU time (s)	# of 2×2-Joins		# of Binas
alu2	10	6	144	1013	1287	10.9	236	61.0	147
alu4	14	8	349	3170	3850	123.9	479	72.9	352
cmb	16	4	25	26	50	7.2	25	100.0	26
f51m	8	8	29	131	177	4.3	64	45.3	34
lal	26	19	54	101	193	25.9	61	88.5	54
term1	34	10	78	202	306	77.3	116	67.2	79
ttt2	24	21	103	274	472	30.8	127	81.1	104
x3	135	99	494	1406	2322	1320.1	548	90.1	494
z4ml	7	4	6	18	12	2.8	12	50.0	12

Joins than the method in [24]. We can synthesize a circuit with only 45.3% of the number of 2×2-Joins in the best case and with 72.9% on average. Furthermore, our method can synthesize circuits with slightly fewer logic elements than the method in [20]. By using our method, we can synthesize small-area dual-rail SFQ circuits. Most of the execution time of our method is expended in reduction of the size of RSBDDs. Once we can obtain reduced RSBDDs, we can synthesize circuits fast.

3.6 Summary of the Chapter

We have proposed a new method of logic synthesis for dual-rail SFQ digital circuits. In the method, we construct a root-shared binary decision diagram (RSBDD) from given logic functions and reduce the size of the constructed RSBDD by a variable re-ordering technique. Then, we construct a dual-rail SFQ circuit using the reduced RSBDD. We have implemented the proposed method and have synthesized some benchmark circuits. The experimental results show that the proposed method can synthesize dual-rail SFQ digital circuits which consist of about 27.1% fewer logic elements than those synthesized by a Transduction-based method.

Using the proposed method, we can design small-area dual-rail SFQ digital circuits. This study is a valuable step for CAD systems using dual-rail representation.

Chapter 4

Clock Scheduling for Synchronous Clocking SFQ Digital Circuits

4.1 Introduction

In this chapter, we propose an algorithm for clock scheduling of concurrent-flow clocking SFQ digital circuits with passive-transmission-lines (PTLs). Now, PTLs are developed as next generation interconnections of SFQ circuits[16–18]. Clock scheduling becomes important when PTLs are used as interconnections. PTLs can transmit pulses at almost the speed of light. However, an active splitter called SPL, which has much larger delay than a PTL, is required for splitting a pulse. The number of SPLs in a path is the key factor of the delay of data and clock pulses, and therefore, adjustment of the number of SPLs is the key factor for the performance (e.g., clock frequency and/or circuit area) of circuits. Furthermore, by combining delay insertion on data paths, clock frequency can become much higher. The amount of delays to be inserted is largely affected by clock scheduling. Therefore, studies of clock scheduling algorithms is important for designing high-performance circuits.

For CMOS digital circuits, several clock scheduling algorithms have been proposed[29, 30]. However, these algorithms are designed for circuits with an assumption that arbitrary delays are achievable. In SFQ circuits, since generation of SFQ pulses by JJs requires a fixed period, delays of paths take discrete values. Therefore, a new algorithm is necessary for SFQ digital circuits.

Given a clock period, the algorithm to be proposed determines the number of SPLs on each clock path and the delay that should be inserted on each data path. By restricting the solution space to be searched, the computation time of the proposed algorithm is polynomial of the number of gates. In addition, the algorithm can also be used to minimize the clock period by carrying out binary search.

Experimental results on smaller benchmark circuits show that the proposed algorithm can obtain the optimum solutions, i.e., the same results with an integer linear programming (ILP) solver. Experimental results on larger benchmark circuits show that the proposed algorithm can obtain near optimal solutions in which inserted delay elements are 59.0% fewer and the height of clock trees are 40.4% shorter on average than those by a straightforward algorithm. The minimum clock periods obtained by the proposed algorithm are 19.0% shorter on average than those by the straightforward algorithm.

The rest of this chapter is organized as follows. In Section 4.2, we describe concurrent-flow clocking SFQ digital circuits. Then, in Section 4.3, we propose an algorithm for clock scheduling. In Section 4.4, we show experimental results. In Section 4.5, we give the summary of this chapter.

4.2 Concurrent-flow Clocking SFQ Digital Circuits

When we construct concurrent-flow clocking SFQ digital circuits, we have to provide clock pulses to all logic gates and have to insert buffers with clock input (D flip-flops) into appropriate positions of data paths for constructing a micropipeline. We show an example of a concurrent-flow clocking circuit in Fig. 4.1. The circuit is a full adder. Clock pulses are provided for all logic gates and buffers, and a micropipeline is constructed. By arranging the timing of clock input, i.e., by utilizing clock skew, we can achieve high clock

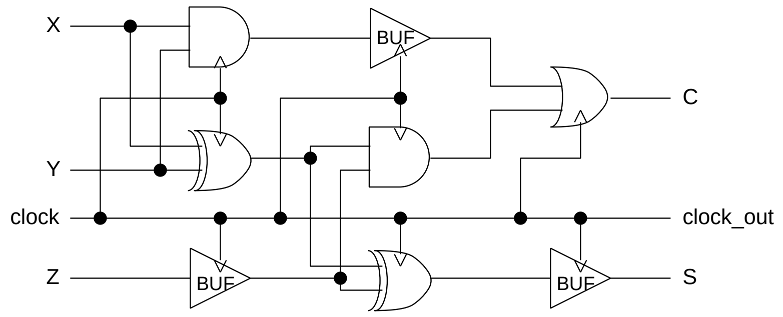


Figure 4.1: An example of a concurrent-flow clocking circuit.

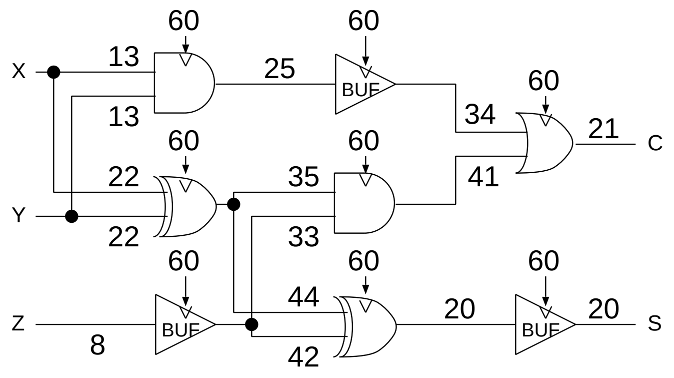
frequency.

In this chapter, we consider the following gates and elements.

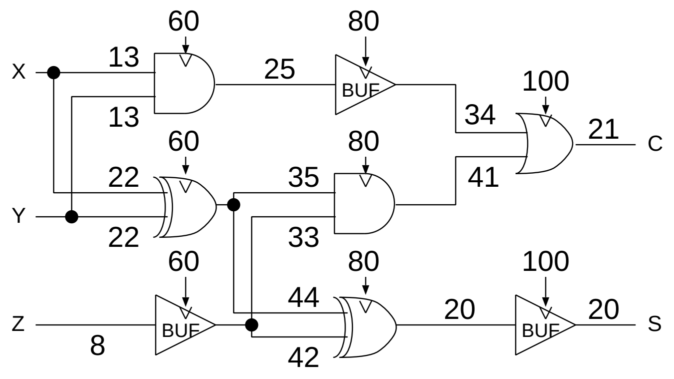
- 2-AND,
- 2-OR,
- 2-XOR,
- NOT,
- BUF (buffer),
- SPL (splitter)[1],
- JTL (Josephson-transmission-line)[1],
- PTL (passive-transmission-line)[16–18].

2-AND, 2-OR and 2-XOR gates are AND, OR and XOR gates with two fan-in, respectively. All logic gates and BUFs are driven by clock pulses. An SPL splits an input pulse to two outputs. JTLs and PTLs are elements for 1-to-1 interconnection.

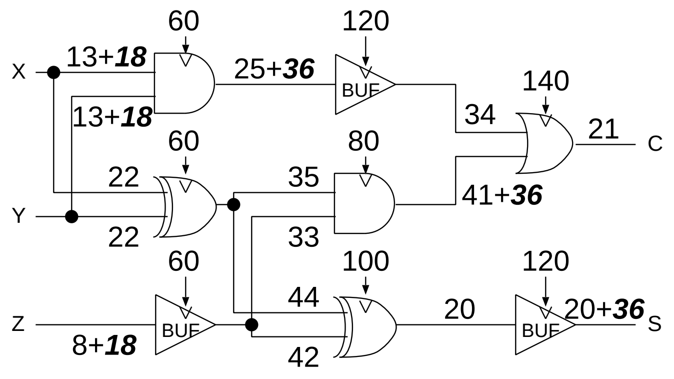
PTLs are transmission lines for SFQ circuits and can transmit pulses very fast. Since SPLs have much larger delay than PTLs, adjustment of the number of SPLs on clock paths, i.e., scheduling the arrival time of clock pulses, is important for achieving high



(a) A circuit with zero-skew clocks. (Clock period: 44 ps)



(b) A circuit with skewed clocks. (Clock period: 24 ps)



(c) A circuit with skewed clocks and inserted delays. (Clock period: 17 ps)

Figure 4.2: An example of clock scheduling.

clock frequency. As an example of clock scheduling, we show three full adders in Fig. 4.2. Each number between gates indicates the data delay between them and each number above an arrow indicates the clock delay between the clock source and the corresponding gate. The unit is picosecond. The clock delay is the product of the number of SPLs and the delay of an SPL. The delay of an SPL is 20 ps. These delays are estimated from the SFQ circuit technology[5]. Figure 4.2 (a) is a full adder using zero-skew clocks. Namely, all clock delays are the same. In this case, the minimum clock period is determined by the largest data delay, and is 44 ps. On the other hand, Fig. 4.2 (b) is a full adder using skewed clocks. By delaying the arrival time of clock pulses toward the output side, the period usable for data transmission becomes longer than the clock period. In this example, by inserting clock skew of 20 ps, the clock period can be shortened to $44 - 20 = 24$ ps.

Not only the adjustment of the number of SPLs on clock paths, but also delay insertion on data paths improves the clock frequency. Improvement of clock frequency using clock skew could be restricted by the occurrence of malfunctions, i.e., double-clocking and zero-clocking[31]. Delay insertion between gates relaxes the restriction. Figure 4.2 (c) shows an example of clock scheduling with delay insertion. Bold oblique numbers indicate the inserted data delays. A JTL of unit length is used as a delay element and its delay is 18 ps in this example. By inserting the JTLs appropriately on data paths, the clock period can be shortened to 17 ps. Thus, appropriate clock scheduling makes clock frequency higher. However, clock scheduling may make circuit area larger and clock delays longer. There is a trade-off between clock frequency and the two mentioned.

4.3 Clock Scheduling for Concurrent-flow Clocking SFQ Digital Circuits

We consider a circuit with no feed-back loop. If a circuit has feed-back loops, we can apply the algorithm to be proposed to the combinational parts (the parts that calculates the next state and the output functions) of the circuit.

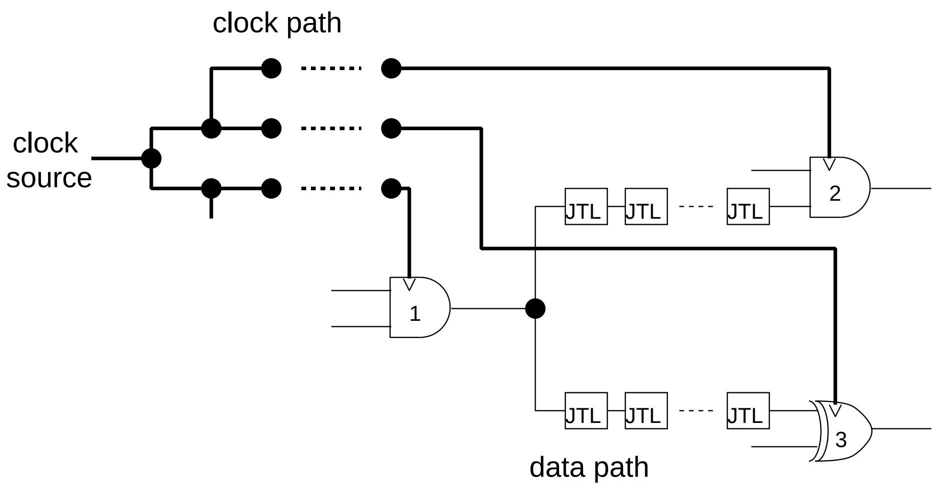


Figure 4.3: A part of a circuit with the assumptions.

4.3.1 Assumptions for Clock Scheduling

The assumptions that we consider for clock scheduling are as follows.

- All logic gates, BUFs, SPLs and JTLs are interconnected with PTLs.
- A JTL of unit length is used for timing adjustment.
- Clock trees consist of PTLs and SPLs.
- The delay of PTLs is 0.

To simplify clock scheduling, we construct clock trees using PTLs and SPLs only. There may be SPLs such that one of the two outputs is not used. Such an SPL is used as a delay element. Since the delay of PTLs is small, we assume that the delay is 0 in the clock scheduling phase and deal with the errors in the placement and routing phase. In addition, we assume that circuits work with ideal conditions. Namely, we do not consider effects of timing jitters, process variation and so on.

We show a part of a circuit with the above assumptions in Fig. 4.3. In the figure, a solid or a bold line indicates a PTL, JTL indicates a JTL of unit length and a black circle indicates an SPL. The JTLs are used for timing adjustment. By determining appropriate number of SPLs on the paths between the clock source and the clock input of gates and appropriate number of JTLs on the paths between adjacent gates driven by clock pulses (in the figure, between gates 1 and 2, and between gates 1 and 3), we can design circuits with higher clock frequency. Hereafter, we call a JTL of unit length 'JTL,' a gate driven by clock pulses 'clocked gate,' a path between the clock source and the clock input of a clocked gate 'clock path' and a path between adjacent clocked gates 'data path.' A clock path consists of SPLs and PTLs, and a data path consists of SPLs, JTLs and PTLs.

4.3.2 Clock Scheduling Problem

We first formulate the clock scheduling problem. The input of the problem is a circuit without clock trees nor JTLs on data paths and a given clock period. The output is the number of SPLs on each clock path and the number of JTLs on each data path. We denote the delay of an SPL by D_{SPL} and the delay of a JTL by D_{JTL}.

Here, we consider an ordered pair of adjacent clocked gates i and j, (i, j), e.g. $(1, 2)$ and $(1, 3)$ in Fig. 4.3. Hereafter, we use the term 'pair' for adjacent clocked gates. We let $t_{cd}(i)$ and $t_{cd}(j)$ be the delays of a clock pulse from the clock source to gates i and j, respectively, $d(i, j)$ be the delay of a data pulse from gate i to gate j, $\delta_H(j)$ and $\delta_S(j)$ be the hold time and the setup time of gate j, respectively, and T_{CP} be the given clock period. Here, $t_{cd}(i)$, $t_{cd}(j)$ and $d(i, j)$ are the values that we want to determine, while $\delta_H(j)$ and $\delta_S(j)$ are the values given by the specification of gates. T_{CP} is the value given by a designer.

We show the timing diagram for correct operation in Fig. 4.4. Double-clocking arises when data from gate i arrives at gate j before $t_{cd}(j) + \delta_H(j)$, while zero-clocking arises when data arrives at gate j after $t_{cd}(j) + T_{CP} - \delta_S(j)$[31]. Therefore, the condition for pair (i, j) to be free from double-clocking is inequality (4.1) and that to be free from

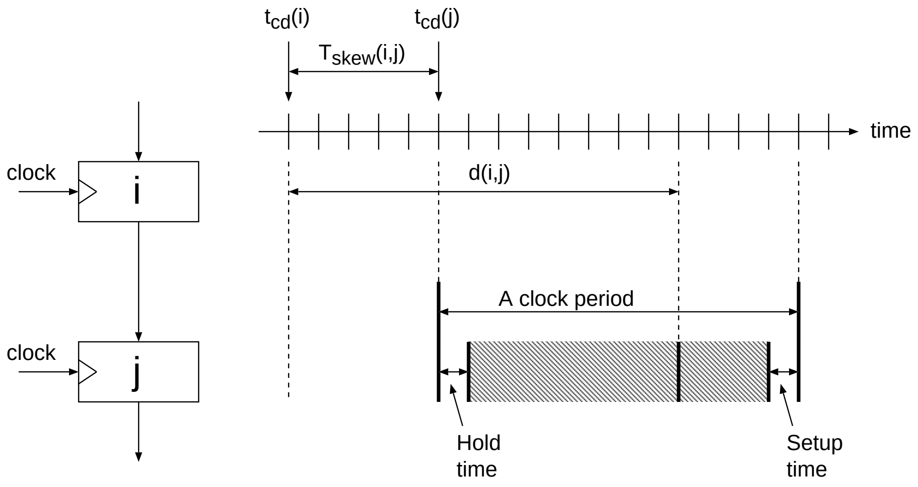


Figure 4.4: Timing diagram of a gate pair.

zero-clocking is inequality (4.2).

$$t_{cd}(i) + d(i,j) \geq t_{cd}(j) + \delta_H(j), \tag{4.1}$$

$$t_{cd}(i) + d(i,j) \leq t_{cd}(j) + T_{CP} - \delta_S(j). \tag{4.2}$$

Let $T_{skew}(i,j) = t_{cd}(i) - t_{cd}(j)$. Since the delay of PTLs is 0, the delay on a clock path is the summation of that of SPLs on it. We let b_i be the number of SPLs on the path from the clock source to gate i. Then $T_{skew} = D_{SPL} \cdot (b_i - b_j)$. Here, b_i and b_j are the values we want to determine.

$d(i,j)$ consists of the delay of gate i, that of the SPLs and that of the JTLs on the path between gates i and j. The delay of gate i and that of SPLs are given by the circuit structure. We let $d_g(i,j)$ be the sum of the delay of gate i and that of SPLs, and $c_{(i,j)}$ be the number of JTLs between gates i and j, then $d(i,j) = d_g(i,j) + D_{JTL} \cdot c_{(i,j)}$. Here, $c_{(i,j)}$ is the value we want to determine. Inequalities (4.1) and (4.2) are rewritten as follows. Note that, b_i, b_j and $c_{(i,j)}$ are nonnegative integers.

$$D_{SPL} \cdot (b_i - b_j) \geq -d_g(i,j) + \delta_H(j) - D_{JTL} \cdot c_{(i,j)}, \tag{4.3}$$

$$D_{SPL} \cdot (b_j - b_i) \quad \geq \quad d_g(i,j) + \delta_S(j) - T_{CP} + D_{JTL} \cdot c_{(i,j)}. \tag{4.4}$$

By calculating b_i's and $c_{(i,j)}$'s that satisfy inequalities (4.3) and (4.4) for all pairs in the circuit, we can determine clock scheduling for the given clock period. Here, D_{SPL}, D_{JTL}, $d_g(i,j)$'s, $\delta_H(j)$'s and $\delta_S(j)$'s are the values given by the specification of gates and elements, and the circuit structure, and T_{CP} is the value given by a designer. b_i's and $c_{(i,j)}$'s are the values we want to determine.

For designing circuits with higher performance, calculating smaller b_i's and $c_{(i,j)}$'s is important. Smaller b_i's make the delay of the circuit shorter and smaller $c_{(i,j)}$'s make the area of the circuit smaller. As a method for solving the clock scheduling problem, formulation of integer linear programming (ILP) exists. By setting $\sum_{n=1}^{\#\text{ of clocked gates}} b_n$ or $b_{\text{the-last-gate}}$ as a cost function, we can solve the clock scheduling problem by ILP solvers. However, as shown in the experimental results later, the computation cost is very high. Therefore, in this chapter, by restricting the solution space to be searched, we solve the problem approximately. First, we describe a straightforward algorithm and then propose an improved algorithm.

4.3.3 Straightforward Algorithm

The fundamental idea is delaying the arrival time of clock pulses for clocked gates in latter stages. Here, the stage of a clocked gate is the number of clocked gates on data paths from the circuit input to it. First, we give a definition for a possible value of $(b_j - b_i)$.

Definition 1: An integer $b_{ij} = b_j - b_i$ is **a possible value** if there is a nonnegative integer $c_{(i,j)}$ such that inequalities (4.3) and (4.4) hold.

We number all clocked gates so that for each pair (i,j), i is smaller than j. We deal with all input ports as a single clocked input gate and all output ports as a single clocked output gate, and number the input gate 0. To restrict the solution space to be searched, we determine an upper bound for the difference of the arrival time of clock pulses between

```
Step 1:
  for each $(i, j)$,
    $A[(i, j)] = \{$possible values of $(b_j - b_i)$ under the given upper bound$\}$;
  $b_0 = 0$;
Step 2:
  for $s = 1$ to max stage do
    $a =$ the minimum value that is common in all $A[(i, j)]$ such that $j$ is in stage $s$;
    for all $(i, j)$ such that $j$ is in stage $s$ do
      $b_j = b_i + a$;
    end for
  end for
Step 3:
  for each $(i, j)$, calculate $c_{(i,j)}$;
```

Figure 4.5: Straightforward algorithm.

adjacent clocked gates, i.e., an upper bound for $D_{SPL} \cdot |b_i - b_j|$, in advance. We show a straightforward algorithm in Fig. 4.5. Here, $A[(i, j)]$ is a set of possible values of $(b_j - b_i)$. If $A[(i, j)]$ becomes an empty set, feasible solution does not exist under the given clock period and the given upper bound. By determining the upper bound, we can obtain A's as finite sets. We set $b_0 = 0$ and calculate other b_i's as the relative values to b_0.

First, we perform initialization in **Step 1**. We calculate possible values of $(b_j - b_i)$ as $A[(i, j)]$. In **Step 2**, for a stage, we select the minimum value that is common in all $A[(i, j)]$ such that j is in the stage and calculate b_i's. Finally, we calculate $c_{(i,j)}$'s in **Step 3**. If there is no common value in **Step 2**, feasible solution does not exist. We show an illustration of the straightforward algorithm in Fig. 4.6.

The straightforward algorithm is a backtrack-free algorithm, and therefore, we can determine clock scheduling fast. However, since all b_i's in a stage are set to the same value, the performance of the solution is not high. In the new algorithm to be proposed, by setting each b_i in a stage independently, we improve the performance.

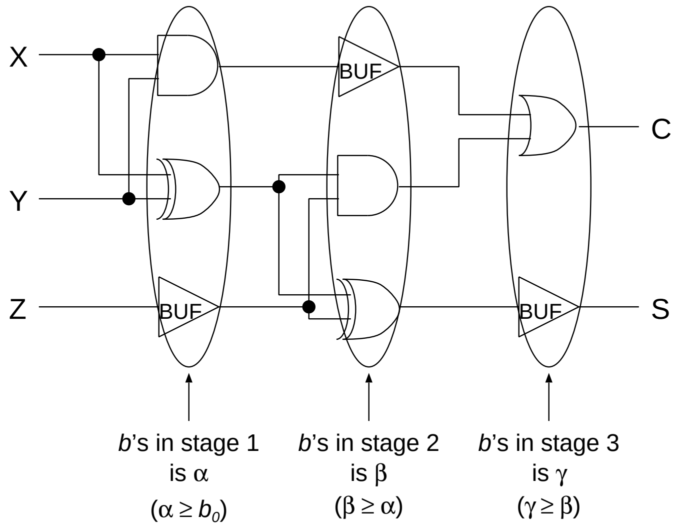

X
Y
Z
BUF
BUF
BUF
C
S
b's in stage 1 is α (α ≥ *b*₀)
b's in stage 2 is β (β ≥ α)
b's in stage 3 is γ (γ ≥ β)

Figure 4.6: An illustration of the straightforward algorithm.

4.3.4 Proposed Algorithm

Since A's may contain non-continuous integers, simple selection of a value from A's does not give a correct solution. Therefore, we calculate sets of candidate values of b_i's and update them sequentially.

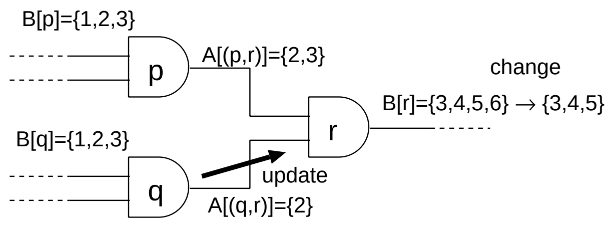
We show the proposed algorithm in Fig. 4.7. We begin the steps with pairs including the input gate. $B[i]$ is a set of candidate values of b_i. In the proposed algorithm, initial B's are calculated using A's under the given upper bound. Then, all candidates of b_i's which do not satisfy inequalities (4.3) and (4.4) are removed. By confirming whether the consequently obtained b_i's and $c_{(i,j)}$'s satisfy inequalities (4.3) and (4.4), the correctness of the obtained solution is guaranteed.

For the given clock period, the proposed algorithm can find a solution when the clock scheduling problem has a solution such that $\max_{(i,j)}\{D_{SPL} \cdot |b_i - b_j|\}$ is less than or equal to the given upper bound, because it searches for the solution from all of the candidates under the given upper bound. By setting the upper bound large enough, the proposed algorithm can find a solution for all problems.

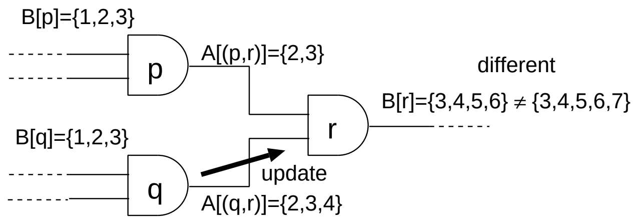
```
Step 1:
  for each $(i, j)$,
    $A[(i, j)] = \{$possible values of $(b_j - b_i)$ under the given upper bound$\}$;
  for each $i$, $B[i] = \emptyset$;
  $B[0] = \{0\}$;
  initialize $Q$ as empty;
Step 2:
  for all $(i, j)$ do
    $candidate\_B = \{x + y \,|\, \forall x, y, x \in B[i], y \in A[(i, j)]\}$;
    if $B[j] == \emptyset$ then
      $B[j] = candidate\_B$;
    else if $B[j] \subset candidate\_B$ then
      $B[j] = B[j] \cap candidate\_B$;
      append pair $((i, j), \leftarrow)$ to $Q$;
    else if $B[j] \supset candidate\_B$ then
      $B[j] = B[j] \cap candidate\_B$;
      for each $(i', j)$ such that $i' < i$,
        append pair $((i', j), \leftarrow)$ to $Q$;
    else if $B[j] \neq candidate\_B$ then
      $B[j] = B[j] \cap candidate\_B$;
      for each $(i', j)$ such that $i' \leq i$,
        append pair $((i', j), \leftarrow)$ to $Q$;
    end if
  end for
Step 3:
  while $Q \neq \emptyset$ do
    select the top element of $Q$ $((i, j), d)$ and remove it;
    if $d == \leftarrow$ then
      $temp\_B = \{x - y \,|\forall x, y, x \in B[j], y \in A[(i, j)]\}$;
      if $B[i] \not\supseteq temp\_B$ then
        $B[i] = B[i] \cap temp\_B$;
        for each $(i, p)$ such that $p \neq j$,
          append $((i, p), \rightarrow)$ to $Q$;
        for each $(q, i)$, append $((q, i), \leftarrow)$ to $Q$;
      end if
    else
      $temp\_B = \{x + y \,|\forall x, y, x \in B[i], y \in A[(i, j)]\}$;
      if $B[j] \not\supseteq temp\_B$ then
        $B[j] = B[j] \cap temp\_B$;
        for each $(q, j)$ such that $q \neq i$,
          append $((q, j), \leftarrow)$ to $Q$;
        for each $(j, p)$, append $((j, p), \rightarrow)$ to $Q$;
      end if
    end if
  end while
Step 4:
  if for $(i, j)$, there are values in $B[i]$ that cannot be produced from $A[(i, j)]$ and the minimum value of $B[j]$ then
    remove the values from $B[i]$;
    for each $(i, p)$, append $((i, p), \rightarrow)$ to $Q$;
    for each $(q, i)$, append $((q, i), \leftarrow)$ to $Q$;
    goto Step 3;
  end if
Step 5:
  for each $i$, $b_i$ = the minimum value in $B[i]$;
Step 6:
  for each $(i, j)$, calculate $c_{(i,j)}$;
```

Figure 4.7: Proposed algorithm.

First, we perform initialization in **Step 1**. Then we produce initial B's from the circuit input to output unidirectionally in **Step 2** and remove all candidates which do not satisfy inequalities (4.3) and (4.4) in **Steps 3** and **4**. Finally, we select b_i's in **Step 5** and calculate $c_{(i,j)}$'s in **Step 6**. We prepare a queue Q for **Steps 3** and **4**. In the queue, we



(a) The values in $B[r]$ change.



(b) $B[r]$ is different from the candidate.

Figure 4.8: Examples requiring **Step 3**.

store pairs of adjacent clocked gates and the direction of the update ($\leftarrow$ or $\rightarrow$). When a pair is (i, j) and the direction is $\leftarrow$, we update $B[i]$ using $B[j]$, while when the direction is $\rightarrow$, we update $B[j]$ using $B[i]$. If $B[i]$ becomes an empty set in the execution of the algorithm, feasible solution does not exist under the given clock period and the given upper bound, and the algorithm terminates.

When $B[i]$ is updated in **Step 2**, candidates which do not satisfy the inequalities may be produced in other B's. **Step 3** is carried out for removing such candidates. Here, we consider two examples. Figure 4.8 shows two intermediate results of **Step 2**. We update $B[r]$ using $B[q]$ and $A[(q,r)]$. In the case of Fig. 4.8 (a), $B[r]$ changes, and therefore, $B[p]$ may change. We append $((p,r), \leftarrow)$ to Q and update $B[p]$ by **Step 3**. On the other hand, in the case of Fig. 4.8 (b), $B[r]$ does not change. However, the candidate of $B[r]$ that is produced from $B[q]$ and $A[(q,r)]$ is different from the updated $B[r]$. This may cause the change of $B[q]$, and therefore, we append $((q,r), \leftarrow)$ to Q and update $B[q]$ by **Step 3**.

For selecting smaller b_i's, we want to select the minimum value in $B[i]$ as b_i. However, A's may contain non-continuous integers and therefore, such selection may cause an incor-

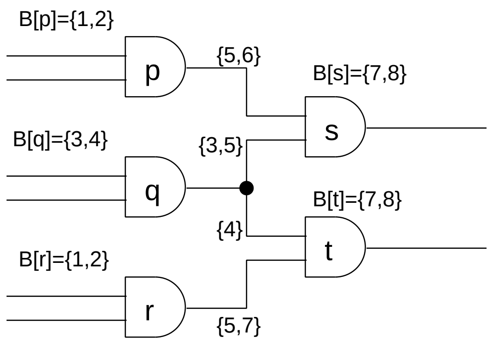


Figure 4.9: An example requiring **Step 4**.

rect solution. We have to remove the values that cannot be produced from the minimum values of B's and A's. Here, we consider an example shown in Fig. 4.9. It is a part of a circuit which is obtained by applying **Steps 1** to **3**. In the circuit, we cannot obtain appropriate values of b_i's by selecting the minimum values of B's from input or output. Here, we consider the case that we select values from output. First, we select 7 for gates s and t. Then, we select values for gates p, q and r. From $A[(q, s)]$, the value for gate q is 4, while from $A[(q, t)]$, the value is 3. For removing this contradiction, we carry out **Step 4**.

Now we consider the computation time of the proposed algorithm. We let the number of clocked gates and the largest stage be g and s, respectively. Since fan-in of each gate is at most two, the number of pairs is $O(g)$. The computation time of **Steps 1**, **5** and **6** is all $O(g)$. The computation time of **Step 2** is $O(g(g + s))$ because that of the production of *candidate_B* is $O(s)$, that of the appending of pairs in a **for**-loop is $O(g)$ and the number of **for**-loops is $O(g)$. Similarly, the computation time of a **while**-loop of **Step 3** is $O(g + s)$. Since the maximum number of elements in all B's is $O(gs)$, the maximum repetitions of **while**-loops required in **Step 3** are $O(gs)$. Therefore, the computation time of **Step 3** is $O(gs(g+s))$. The computation time of **Step 4** is also $O(gs(g+s))$. Since the largest stage is smaller than the number of gates, i.e., $s < g$, the computation time of the proposed algorithm is $O(g^3)$.

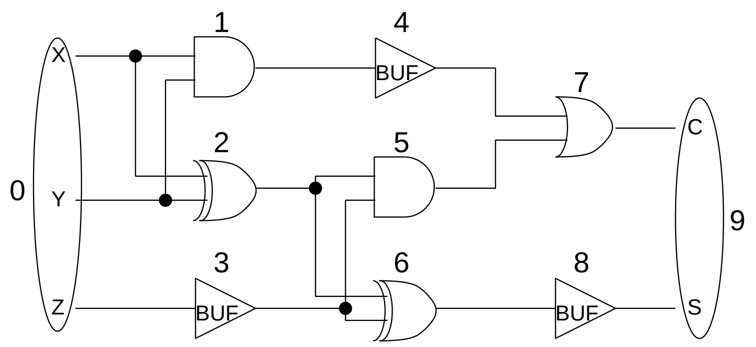


Figure 4.10: A full adder.

Table 4.1: The parameters of gates and elements (ps).

	delay	δ_H	δ_S
BUF	20	2	-2
AND	27	8	-7
XOR	22	-1	2
OR	21	-1	14
NOT	22	10	19
SPL	20	-	-
JTL	18	-	-

We can obtain the minimum clock period using the proposed algorithm. For simplicity, we assume that for all pairs, $-d_g(i,j) + \delta_H(j) \leq 0$. By adding delays to $d_g(i,j)$, we can always satisfy the assumption. When we set the clock period to $\max_{(i,j)}\{d_g(i,j) + \delta_S(j)\}$, the circuit always works correctly. Therefore, by carrying out binary search on a clock period, we can determine the minimum clock period. We set the maximum value as an initial value and search for the minimum clock period that is schedulable. The number of SPLs on data paths between adjacent clocked gates is at most the number of gates and the maximum delay of data pulses is $O(g)$. Therefore, the computation time of calculating the minimum clock period is $O(g^4)$.

4.3.5 An Example: Clock Scheduling of a Full Adder

Here, as an example, we consider clock scheduling of a full adder shown in Fig. 4.10. We show the delay, hold time (δ_H) and setup time (δ_S) of the gates in Table 4.1[5]. $D_{SPL} = 20$ and $D_{JTL} = 18$.

We introduce an input gate for all input ports (gate number 0) and an output gate for all output ports (gate number 9). We set $T_{CP} = 14$ and the upper bound of $D_{SPL} \cdot |b_i - b_j|$ to $5D_{SPL}$. The following inequalities are obtained.

$$\begin{aligned}
(0,1): \quad 20(b_0 - b_1) &\geq -20 + 8 - 18c_{(0,1)},\\
20(b_1 - b_0) &\geq 20 - 7 - 14 + 18c_{(0,1)},\\
(0,2): \quad 20(b_0 - b_2) &\geq -20 - 1 - 18c_{(0,2)},\\
20(b_2 - b_0) &\geq 20 + 2 - 14 + 18c_{(0,2)},\\
(0,3): \quad 20(b_0 - b_3) &\geq 0 + 2 - 18c_{(0,3)},\\
20(b_3 - b_0) &\geq 0 - 2 - 14 + 18c_{(0,3)},\\
(1,4): \quad 20(b_1 - b_4) &\geq -27 + 2 - 18c_{(1,4)},\\
20(b_4 - b_1) &\geq 27 - 2 - 14 + 18c_{(1,4)},\\
(2,5): \quad 20(b_2 - b_5) &\geq -42 + 8 - 18c_{(2,5)},\\
20(b_5 - b_2) &\geq 42 - 7 - 14 + 18c_{(2,5)},\\
(2,6): \quad 20(b_2 - b_6) &\geq -42 - 1 - 18c_{(2,6)},\\
20(b_6 - b_2) &\geq 42 + 2 - 14 + 18c_{(2,6)},\\
(3,5): \quad 20(b_3 - b_5) &\geq -40 + 8 - 18c_{(3,5)},\\
20(b_5 - b_3) &\geq 40 - 7 - 14 + 18c_{(3,5)},\\
(3,6): \quad 20(b_3 - b_6) &\geq -40 - 1 - 18c_{(3,6)},\\
20(b_6 - b_3) &\geq 40 + 2 - 14 + 18c_{(3,6)},\\
(4,7): \quad 20(b_4 - b_7) &\geq -20 - 1 - 18c_{(4,7)},
\end{aligned}$$

$$\begin{aligned}
& 20(b_7-b_4) && \geq && 20+14-14+18c_{(4,7)}, \\
(5,7): \ & 20(b_5-b_7) && \geq && -27-1-18c_{(5,7)}, \\
& 20(b_7-b_5) && \geq && 27+14-14+18c_{(5,7)}, \\
(6,8): \ & 20(b_6-b_8) && \geq && -22+2-18c_{(6,8)}, \\
& 20(b_8-b_6) && \geq && 22-2-14+18c_{(6,8)}, \\
(7,9): \ & 20(b_7-b_9) && \geq && -21-18c_{(7,9)}, \\
& 20(b_9-b_7) && \geq && 21-14+18c_{(7,9)}, \\
(8,9): \ & 20(b_8-b_9) && \geq && -20-18c_{(8,9)}, \\
& 20(b_9-b_8) && \geq && 20-14+18c_{(8,9)}.
\end{aligned}$$

We apply the proposed algorithm to the inequalities. The following A's are obtained.

$$\begin{aligned}
A[(0,1)] &= \{0,1,2,3,4,5\}, \\
A[(0,2)] &= \{1,4,5\}, \\
A[(0,3)] &= \{1,2,3,4,5\}, \\
A[(1,4)] &= \{1,2,3\}, \\
A[(2,5)] &= \{2,3,4,5\}, \\
A[(2,6)] &= \{2,3\}, \\
A[(3,5)] &= \{1,2,3,4,5\}, \\
A[(3,6)] &= \{2,5\}, \\
A[(4,7)] &= \{1\}, \\
A[(5,7)] &= \{5\}, \\
A[(6,8)] &= \{1,3,4,5\}, \\
A[(7,9)] &= \{1,4,5\},
\end{aligned}$$

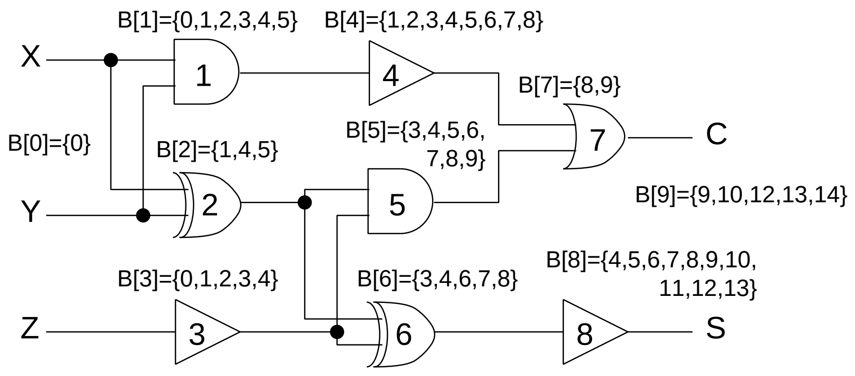
$$A[(8,9)] \quad = \quad \{1,3,4,5\}.$$

As an example, we consider a calculation of $A[(0,2)]$. In this example, the given predetermined upper bound is $5D_{SPL}$ and therefore, the allowed $|b_0 - b_2|$ is less than or equal to 5. In this condition, we search for possible values of $(b_2 - b_0)$. When $(b_2 - b_0) = 1$, inequalities about $(0,2)$ hold with $c_{(0,2)} = 0$. However, when $(b_2 - b_0) = 2$, no $c_{(0,2)}$ exists such that the inequalities hold. We check other $(b_2 - b_0)$'s in the same way and obtain $A[(0,2)] = \{1,4,5\}$.

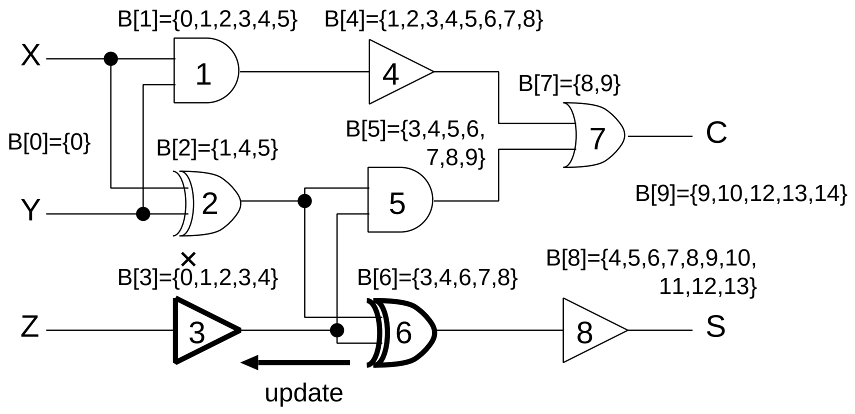
Consequently, the initial B's are obtained from A's by **Step 2**. We show the values in Fig. 4.11(a).

$$\begin{aligned}
B[1] &= \{0,1,2,3,4,5\},\\
B[2] &= \{1,4,5\},\\
B[3] &= \{1,2,3,4,5\},\\
B[4] &= \{1,2,3,4,5,6,7,8\},\\
B[5] &= \{3,4,5,6,7,8,9,10\},\\
B[6] &= \{3,4,6,7,8\},\\
B[7] &= \{8,9\},\\
B[8] &= \{4,5,6,7,8,9,10,11,12,13\},\\
B[9] &= \{9,10,12,13,14\}.
\end{aligned}$$

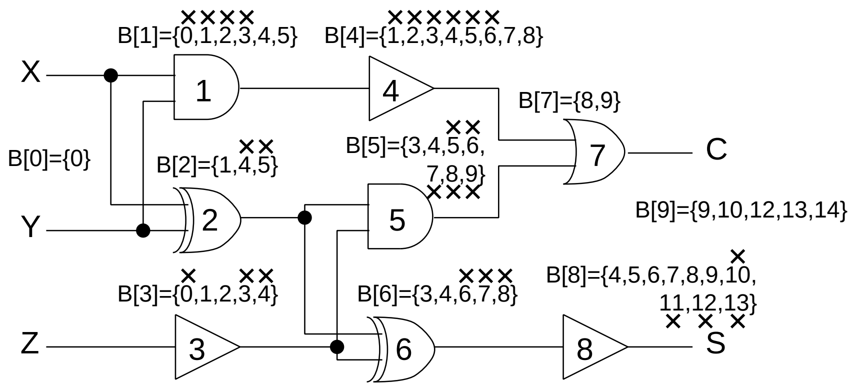
As an example, we consider a calculation of $B[7]$. First, we calculate $B[7]$ by $B[4]$ and $A[(4,7)]$, and then, we update $B[7]$ by $B[5]$ and $A[(5,7)]$. *candidate_B* obtained from $B[4]$ and $A[(4,7)]$ is $\{x + y \,|\, \forall x, y, x \in B[4], y \in A[(4,7)]\} = \{2,3,4,5,6,7,8,9\}$. Since the first $B[7]$ is $\emptyset$, $B[7] = \textit{candidate\_B} = \{2,3,4,5,6,7,8,9\}$. *candidate_B* obtained from $B[5]$ and $A[(5,7)]$ is $\{x + y \,|\, \forall x, y, x \in B[5], y \in A[(5,7)]\} = \{8,9,10,11,12,13,14,15\}$. Since $B[7] \neq \textit{candidate\_B}$, a new $B[7] = B[7] \cap \textit{candidate\_B} = \{2,3,4,5,6,7,8,9\} \cap$



(a) Initially obtained B's from A's.



(b) Process of $((4, 7), \leftarrow)$.



(c) Final values.

Figure 4.11: Application for a full adder.

$\{8, 9, 10, 11, 12, 13, 14, 15\} = \{8, 9\}$. Here, $((4, 7), \leftarrow)$ and $((5, 7), \leftarrow)$ are appended to the queue Q.

Finally, in Q, there are elements:

$$((3, 5), \leftarrow), ((3, 6), \leftarrow), ((4, 7), \leftarrow), ((5, 7), \leftarrow), ((8, 9), \leftarrow).$$

We update B's using the elements in Q. As an example, we consider $((4, 7), \leftarrow)$. We update $B[4]$ using $B[7]$ and $A[(4, 7)]$ because the direction is "$\leftarrow$." Since $\{x - y \mid \forall x, y, x \in B[7], y \in A[(4, 7)]\} = \{7, 8\}$, and intersection of the values and the original $B[4]$, i.e., $\{7, 8\} \cap \{1, 2, 3, 4, 5, 6, 7, 8\}$ is $\{7, 8\}$, the new $B[4]$ is $\{7, 8\}$. Here, $((1, 4), \leftarrow)$ is appended to Q. We show the values after this process in Fig. 4.11(b).

When we repeat these processes until Q becomes empty, we obtain the values shown in Fig. 4.11(c). By selecting the minimum values in B's, we obtain $b_1 = 4$, $b_2 = 1$, $b_3 = 1$, $b_4 = 7$, $b_5 = 3$, $b_6 = 3$, $b_7 = 8$, $b_8 = 4$ and $b_9 = 9$. By calculating $c_{(i,j)}$'s, we obtain $c_{(0,1)} = 4$, $c_{(0,2)} = 0$, $c_{(0,3)} = 2$, $c_{(1,4)} = 2$, $c_{(2,5)} = 1$, $c_{(2,6)} = 0$, $c_{(3,5)} = 1$, $c_{(3,6)} = 0$, $c_{(4,7)} = 0$, $c_{(5,7)} = 4$, $c_{(6,8)} = 0$, $c_{(7,9)} = 0$ and $c_{(8,9)} = 5$.

4.4 Evaluation of the Proposed Algorithm

We have implemented the proposed algorithm by using the C programming language and have applied it to clock scheduling of several benchmark circuits in LGSynth'91[28]. The experimental environment is a workstation with Itanium2 1.6 GHz CPU and 8 GByte memory. We have used the parameters shown in Table 4.1.

4.4.1 Comparison with an ILP Solver

We have compared the proposed algorithm with lp_solve which is an ILP solver[32]. We have used $\sum_{n=1}^{\#\text{ of clocked gates}} b_n$ as the cost function and have set the clock period to 50 ps. For the proposed algorithm, we have set the upper bound of $D_{SPL} \cdot |b_i - b_j|$ to $10D_{SPL}$. We have calculated the number of SPLs in each clock tree, the number of inserted JTLs

Table 4.2: A comparison of the proposed algorithm with an ILP solver.

	# of gates	# of pairs	# of stages	max. delay (ps)	proposed algorithm # of SPLs	# of JTLs	height of c. t.	exe. time (s)	ILP solver exe. time (s)
C17	26	32	6	60	27	2	7	0.001	0.005
b1	61	77	8	89	65	0	8	0.001	0.009
cc	338	433	8	109	344	124	15	0.001	122.007
cm138a	92	124	6	95	94	36	9	0.002	0.573
cm150a	258	333	18	109	264	138	13	0.002	50.487
cm151a	128	164	16	89	131	31	10	0.002	1.052
cm152a	89	119	10	89	95	5	9	0.002	0.045
cm162a	294	352	14	106	298	88	14	0.002	53.472
cm163a	272	328	13	106	278	39	11	0.002	6.549
cm42a	107	137	7	106	108	57	10	0.001	2.612
cm82a	72	95	8	80	73	4	8	0.002	0.024
cm85a	171	216	13	86	180	22	13	< 0.000	1.590
cmb	286	343	19	89	288	24	11	0.002	4.776
comp	417	563	20	86	419	51	13	0.003	16.879
cu	301	385	13	109	307	55	13	0.002	7.035
decod	95	159	4	109	101	34	10	0.001	1.726
ldd	468	627	14	109	473	244	14	0.004	130.075
majority	54	70	9	89	58	0	8	< 0.000	0.008
mux	535	670	23	109	543	268	15	0.005	738.279
parity	107	151	12	60	109	0	8	0.002	0.015
pm1	251	329	9	109	258	83	13	0.002	25.807
tcon	82	136	3	109	85	40	9	0.002	1.381
unreg	298	424	7	129	300	239	11	0.003	140.446
x2	187	254	9	109	192	92	11	0.002	17.080

and the height of each clock tree from the results of the execution. We have constructed the clock trees so that the height is the minimum.

We show the experimental results in Table 4.2. "# of gates" is the number of clocked gates including BUFs that are inserted for constructing a micropipeline and “max. delay” is $\max_{(i,j)}\{d_g(i,j) + \delta_S(j)\}$. If we design circuits with zero-skew clocks, the maximum value is the minimum clock period. “height of c. t.” is the height of clock tree. “exe. time” is the user CPU time of the execution.

The experimental results show that for a given clock period, the proposed algorithm can obtain the same results with the ILP solver. The execution time of the proposed algorithm is much shorter than that of the ILP solver.

4.4.2 Comparison with the Straightforward Algorithm

We have compared the proposed algorithm with the straightforward algorithm described in Section 4.3.3. We have set the clock period to 50 ps and have set the upper bound of $D_{SPL} \cdot |b_i - b_j|$ to $10D_{SPL}$. We show the specification of the benchmark circuits that are used for the comparison in Table 4.3 and the experimental results in Table 4.4. The experimental results show that for a given clock period, the proposed algorithm can construct circuits with 59.0% fewer JTLs on average and 40.4% shorter clock trees on average than those by the straightforward algorithm. The number of SPLs required for constructing clock trees by the proposed algorithm is comparable with that by the straightforward algorithm.

We can also use the straightforward algorithm to minimize the clock period in the same way as the proposed algorithm. We have calculated the minimum clock periods. Table 4.5 shows the results that are obtained by the algorithms when we change upper bounds of $D_{SPL} \cdot |b_i - b_j|$. “u. b.” indicates upper bound and “c. p.” indicates clock period. The experimental results show that clock periods have been 6.7% shorter when we have changed the upper bound $10D_{SPL}$ to $30D_{SPL}$ and have been the same when we have changed it $30D_{SPL}$ to $50D_{SPL}$. From the results, we can see that by using only a small value, e.g., $10D_{SPL}$ for the upper bound, we can obtain good clock periods. Clock periods have been 19.0% shorter on average than those by the straightforward algorithm.

Table 4.3: Specification of the benchmark circuits.

	# of gates	# of pairs	# of stages	maximum delay (ps)
9symml	637	870	20	129
C1355	5142	5690	46	121
C432	2788	3005	55	141
C5315	22230	22430	72	149
C6288	54702	57116	242	86
C7552	20664	23402	66	189
alu4	9817	10989	75	169
apex6	3590	4353	19	149
b9	758	916	16	109
cht	842	1214	10	169
count	1396	1537	36	129
des	41551	48525	45	226
f51m	1233	1550	26	149
frg2	14507	16973	30	189
k2	31100	33789	196	189
lal	891	1095	17	109
my_adder	3434	3705	52	100
rot	14631	16022	57	160
sct	925	1134	17	129
term1	4140	4998	32	149
too_large	306909	321400	544	249
ttt2	3593	4264	39	149
vda	5624	6961	56	169
x1	16949	19095	100	169
x3	8305	9886	32	169
z4ml	997	1247	31	129

4.5 Summary of the Chapter

We have proposed an algorithm for clock scheduling of concurrent-flow clocking SFQ digital circuits with PTLs. By using the algorithm, we can determine clock scheduling

Table 4.4: A comparison of the proposed algorithm with the straightforward algorithm.

	proposed algorithm				straightforward algorithm			
	# of SPLs	# of JTLs	height of c. t.	exe. time (s)	# of SPLs	# of JTLs	height of c. t.	exe. time (s)
9symml	645	360	16	0.007	642	801	17	0.007
C1355	5159	2270	30	0.043	5162	1914	32	0.028
C432	2798	566	19	0.026	2804	1228	36	0.016
C5315	22244	9497	31	0.232	22282	42066	105	0.117
C6288	54719	6019	35	1.174	54817	48456	230	0.285
C7552	20679	14139	34	0.215	20719	45934	113	0.116
alu4	9837	12739	48	0.108	9869	18898	96	0.055
apex6	3598	2298	21	0.026	3603	7083	32	0.020
b9	762	169	15	0.005	768	987	23	0.004
cht	845	591	14	0.007	848	1417	15	0.006
count	1410	1085	33	0.012	1417	1466	39	0.005
des	41560	36632	31	0.389	41603	159955	114	0.249
f51m	1242	881	20	0.012	1239	1391	18	0.009
frg2	14525	13734	32	0.115	14557	64977	105	0.080
k2	31123	22455	42	0.611	31223	56177	217	0.176
lal	897	131	15	0.008	904	1079	26	0.005
my_adder	3455	2385	44	0.028	3459	2970	55	0.020
rot	14652	9924	40	0.142	14688	36368	112	0.080
sct	927	166	15	0.009	932	809	16	0.006
term1	4149	2085	22	0.037	4176	9910	63	0.026
too_large	306943	228086	83	13.602	306959	534403	119	1.741
ttt2	3608	2578	26	0.026	3609	6814	38	0.020
vda	5632	8208	30	0.060	5683	14977	122	0.029
x1	16959	4473	29	0.210	16970	25607	40	0.094
x3	8315	4338	23	0.064	8328	25072	59	0.053
z4ml	1008	733	19	0.006	1006	925	17	0.008

for a given clock period in the computation time of $O(g^3)$ where g is the number of clocked gates. Experimental results on smaller benchmark circuits show that the proposed algorithm can obtain the optimum solutions, i.e., the same results with an ILP solver. Experimental results on larger benchmark circuits show that for a given clock period, the proposed algorithm can obtain near optimal solutions in which inserted delay elements have been 59.0% fewer and clock trees have been 40.4% shorter on average than those by a straightforward algorithm. The proposed algorithm can also be used to minimize the clock period. The clock periods obtained by the proposed algorithm have been 19.0% shorter on average than those by the straightforward algorithm.

Using the proposed algorithm, we can design high-throughput concurrent-flow clocking SFQ digital circuits with fewer delay elements. In this chapter, we have assumed that circuits work with ideal conditions and have set the delay of PTLs is 0. Evaluation of timing jitters and errors of the delay of PTLs is left for future works.

Table 4.5: The minimum clock periods.

	proposed algorithm						straightforward algorithm	
	u. b. = $10D_{SPL}$		u. b. = $30D_{SPL}$		u. b. = $50D_{SPL}$		u. b. = $10D_{SPL}$	
	minimum c. p. (ps)	exe. time (s)	minimum c. p. (ps)	exe. time (s)	minimum c. p. (ps)	exe. time (s)	minimum c. p. (ps)	exe. time (s)
9symml	29	0.013	29	0.062	29	0.156	37	0.003
C1355	30	0.098	30	0.551	30	1.377	32	0.030
C432	31	0.102	30	0.520	30	1.351	36	0.017
C5315	31	1.186	30	5.628	30	14.783	41	0.117
C6288	30	5.894	30	41.463	30	111.509	34	0.317
C7552	34	0.675	30	7.195	30	18.874	44	0.120
alu4	34	0.453	30	3.085	30	8.199	41	0.060
apex6	30	0.069	30	0.318	30	0.773	39	0.023
b9	30	0.015	30	0.067	30	0.179	38	0.006
cht	29	0.013	29	0.058	29	0.124	41	0.004
count	30	0.030	30	0.157	30	0.397	39	0.011
des	44	1.560	30	8.532	30	21.855	45	0.249
f51m	29	0.031	29	0.162	29	0.416	39	0.008
frg2	40	0.405	30	2.540	30	6.620	46	0.089
k2	29	3.692	29	24.931	29	66.034	45	0.176
lal	30	0.017	30	0.088	30	0.224	38	0.006
my_adder	30	0.113	30	0.705	30	1.893	30	0.019
rot	36	0.500	30	2.071	30	5.321	41	0.078
sct	30	0.013	30	0.061	30	0.153	40	0.006
term1	34	0.116	30	0.522	30	1.324	42	0.020
too_large	49	100.271	29	542.180	29	1447.116	49	1.923
ttt2	32	0.095	30	0.529	30	1.396	42	0.022
vda	29	0.196	29	1.268	29	3.360	41	0.031
x1	29	0.828	29	5.909	29	15.534	41	0.101
x3	34	0.228	30	1.218	30	3.093	44	0.051
z4ml	29	0.021	29	0.127	29	0.329	37	0.006

Chapter 5

Sequential Circuit Synthesis for Synchronous Clocking SFQ Digital Circuits

5.1 Introduction

In this chapter, we propose a new synthesis method of sequential circuits (circuits with feedback loops) to achieve high-throughput SFQ sequential circuits. When we design an SFQ sequential circuit using synchronous clocking representation, we have to provide two kinds of clock signals of different periods. One is for the combinational blocks (the blocks calculating the next state and the output functions), and the other is for the registers of state variables. The throughput of such an SFQ sequential circuit is determined by the period of the latter kind of clock signal that is usually much longer than the former and the power of high-throughput computation of an SFQ digital circuit is spoilt. Therefore, sequential circuits have been designed by transforming feedback loops smaller or removing feedback loops from the circuits[14, 15]. However, these methods are circuit-specific and cannot be applied to arbitrary types of sequential machines. For developing CAD systems, a synthesis method of sequential circuits which is applicable for arbitrary types of sequential machines is important.

Using the new method to be proposed, we can construct an SFQ sequential circuits without clocked gates in its feedback loops. In the method, we use a 'state module' consisting of a D flip-flop (DFF) and several AND gates. First, we encode states of a sequential machine by one-hot encoding and assign state modules to the states one by one, and then, connect the modules with each other according to the state transition. For the connection, we use CBs, i.e., merger gates without clock signals.

We have implemented the proposed method and have synthesized several benchmark circuits. The experimental results show that compared with a conventional method for CMOS digital circuits, the proposed method synthesizes circuits that work with 4.9 times higher clock frequency and have 17.3% more gates on average.

The rest of this chapter is organized as follows. In Section 5.2, we describe SFQ sequential circuits. Then, in Section 5.3, we propose a method of sequential circuit synthesis. In Section 5.4, we show experimental results. Finally in Section 5.5, we give the summary of this chapter.

5.2 SFQ Sequential Circuits Using Synchronous Clocking Representation

In general, a sequential circuit has feedback loops and registers. The registers store state variables. Since synchronizing clocks are necessary for all logic gates in a synchronous clocking SFQ digital circuit, two kinds of clock signals are necessary. One is for combinational blocks and the other is for registers of state variables. Hereafter, we call the clock signal that drives combinational blocks the 'local clock' and the one that drives registers of state variables the 'system clock.'

In sequential circuit synthesis, there are two methods of state assignment. One is binary encoding and the other is one-hot encoding. In the former method, n states are assigned to $\lceil \log_2 n \rceil$-bit registers. Note that $\lceil \log_2 n \rceil$ is the minimum number of bits to

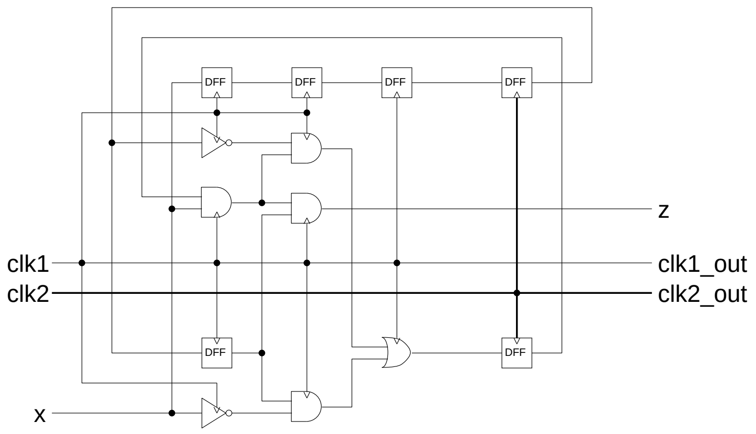


Figure 5.1: An example of an SFQ sequential circuit with binary encoding.

represent n values. In the latter method, n-bit registers are assigned to n states. Namely, one bit is assigned to each state.

We show an example of an SFQ sequential circuit synthesized by a method for CMOS digital circuits with binary encoding in Fig. 5.1. In the circuit, *clk1* is the local clock and *clk2* is the system clock. The throughput of the circuit is limited by the clock period of *clk2* which is much longer than that of *clk1*. By removing clocked gates, i.e., gates driven by a clock signal, from feedback loops, we can drive all clocked gates in the combinational blocks and the registers of state variables by a single clock signal and achieve a high-throughput circuit.

5.3 Method for Synthesizing SFQ Sequential Circuit Using One-hot Encoding

5.3.1 Flow of Sequential Circuit Synthesis

We propose a method of sequential circuit synthesis for SFQ digital circuits. We show the flow of the proposed method in Fig. 5.2. In the method, first, we encode states of a sequential machine by one-hot encoding, and then, assign a 'state module,' which will be described in the next subsection, to each state. Finally, we connect the modules with each

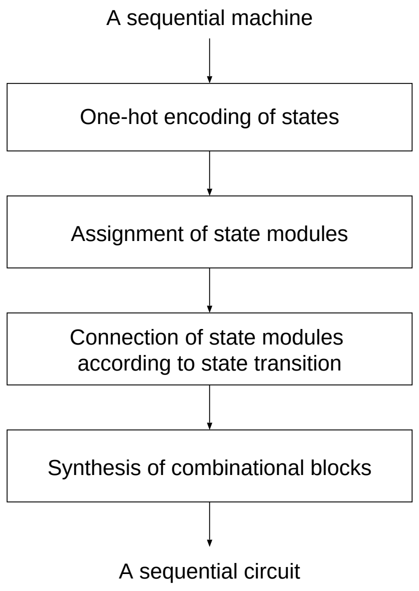

A sequential machine
One-hot encoding of states
Assignment of state modules
Connection of state modules according to state transition
Synthesis of combinational blocks
A sequential circuit

Figure 5.2: Flow of the sequential circuit synthesis.

other according to the state transition and synthesize the combinational blocks, i.e., the blocks calculating the next state and the output functions. For the connection, we use CBs, i.e., merger gates without clock signals. In the synthesis of combinational blocks, we insert buffers with clock input (DFFs) into appropriate positions for timing adjustment. One-hot encoding and synthesis of combinational blocks can be performed in the same way as design of CMOS digital circuits. Since CBs are merger gates without clock signals and state modules are driven by a clock signal, a sequential circuit synthesized by the proposed method has no clocked gates in its feedback loops. In the next subsections, we describe assignment of state modules and connection of them.

5.3.2 Assignment of State Modules

We assign a ‘state module’ shown in Fig. 5.3 to each state of a sequential machine. A state module consists of a DFF and n AND gates. Here, n is the number of next states of the considered state. For example, when the state has three next states, $n = 3$. The

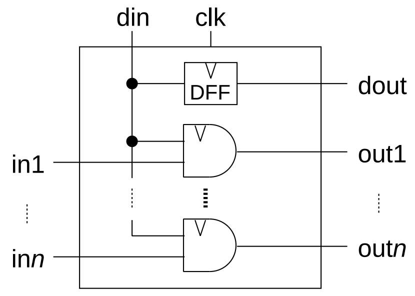


Figure 5.3: State module.

module has *din*, *clk* and *in*1, ..., *inn* as input ports and *dout* and *out*1, ..., *outn* as output ports. *in*1, ..., *inn* are used for construction of the block calculating the next state functions, *dout* is used for construction of the block calculating the output functions and *din* and *out*1, ... *outn* are used for connection of state modules.

Now, we consider an assignment of module M_i to state S_i. When a pulse is input to *din* of M_i, the current state is S_i. A clock pulse is input to *clk* after data pulses are input to *din* and *ink*, then a pulse is output from *outk*. Therefore, if the current state is S_i, a pulse is output from one of *out*1, ..., *outn* of M_i, while if the current state is not S_i, no pulse is output from the output ports of M_i. Since each AND gate has the function of a latch, DFFs for storing the state variables are not necessary.

5.3.3 Connection of State Modules

We connect the assigned state modules with each other according to the state transition. We use CBs for the connection. Here, we consider the state transition from state S_i to S_j by a signal corresponding to an input event. We input a pulse, which is generated by the signal corresponding to the event, to one of *in*'s of M_i and connect its corresponding *out* to *din* of M_j using CBs. In each state transition, there is a pulse in *din* of only one module, and therefore, at most one pulse is input to each CB connecting to modules at a time.

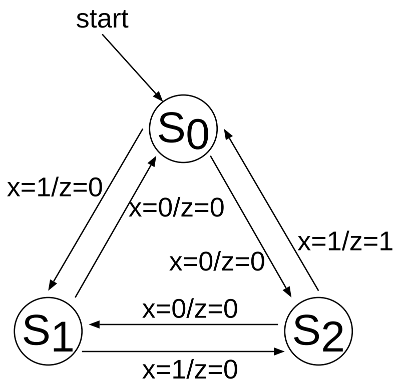


Figure 5.4: An example: state transition diagram.

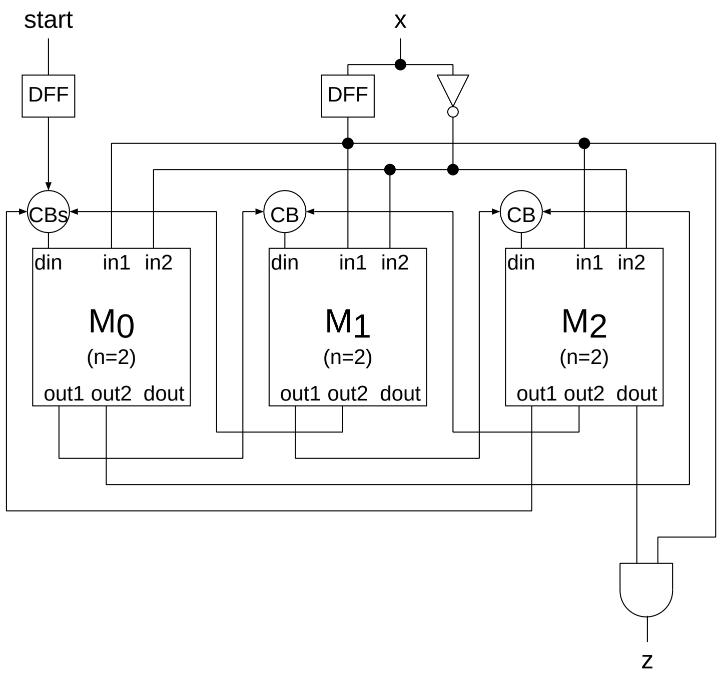


Figure 5.5: An example: synthesized circuit.

5.3.4 An Example: Synthesis of A Sequential Circuit

As an example of synthesis of a sequential circuit, we consider the sequential machine represented by the state transition diagram shown in Fig. 5.4. The machine has one input variable x, one output variable z and three states S_0, S_1 and S_2. The state pointed by an arrow with "start" indicates the initial state. We show the synthesized circuit by the proposed method in Fig. 5.5. M_0, M_1 and M_2 are state modules and correspond to states S_0, S_1 and S_2, respectively.

Now, we consider state S_0 and its corresponding module M_0. Since S_0 has two next states, i.e., S_1 and S_2, the number of *in* or *out* ports, i.e., n of M_0 is two. We consider the state transition from S_0 to S_2. The transition occurs when the input event is $x = 0$, i.e., the input variable x is 0. We connect input x to *in*2 via a NOT gate, that is, we input a pulse to *in*2 when $x = 0$. We connect its output, *out*2, to *din* of M_2 using a CB because the state transition is S_0 to S_2. We use input *start* for the input of the initial pulse. Output z becomes 1 when the state is S_2 and x is 1. Therefore, we construct the output block of the circuit by one AND gate. We insert buffers with clock input (DFFs) to appropriate positions for timing adjustment.

5.3.5 Construction of State Modules

In the design of SFQ digital circuits, layout of a circuit is obtained by placing and routing optimized parts called cells[5]. For achieving a sequential circuit with shorter clock periods, it is effective that we prepare a state module and CBs connecting to its *din* as an optimized cell. However, preparing a large number of optimized cells may be difficult. By preparing several optimized cells as basic cells and constructing state modules with the basic cells, we can construct cells which are almost optimal with respect to clock periods. Now, we consider a basic cell for a state, which has k_m previous states and k_n next states, shown in Fig. 5.6. We construct a state module for a state, which has m previous states and n next states, using the basic cell.

A case of $m > k_m$ and $n \leq k_n$

For convenience, we assume that $m \leq 2k_m$. We can construct a state module for the state using two basic cells as shown in Fig. 5.7. Since there is a pulse in either *din* of

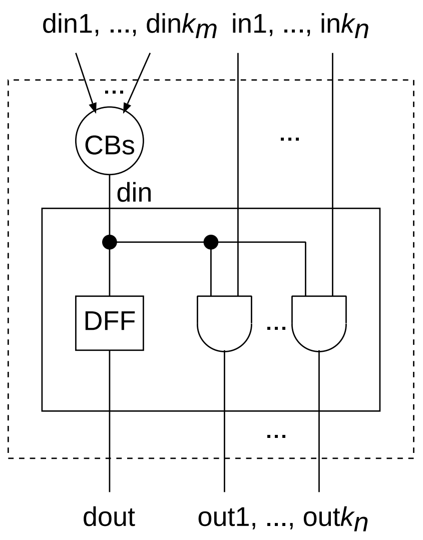


Figure 5.6: A basic cell.

two basic cells, we merge *dout*'s and *out*'s of the two basic cells by CBs. Compared with the optimized cell designed for the state, extra CBs are needed for merging *dout*'s and *out*'s. On the other hand, the number of CBs on the paths connecting to *din* decreases by one. Therefore, the total number of CBs on the paths from the input to the output of the module does not change. Splitters (SPLs) which are indicated by black circles in the figure are necessary for splitting $in1$, …, inn, and therefore, the delay of SPLs for $in1$, …, inn and the interconnections for the SPLs and the CBs merging *dout*'s and *out*'s increases.

When $m > 2k_m$, we can construct the state module using three or more basic cells in the same way.

A case of $m \leq k_m$ and $n > k_n$

For convenience, we assume that $n \leq 2k_n$. We can construct a state module for the state using two basic cells as shown in Fig. 5.8. We split signal lines connecting to *din* using SPLs. Since there are pulses in both *din*'s of the basic cells, *dout* of one basic cell is not

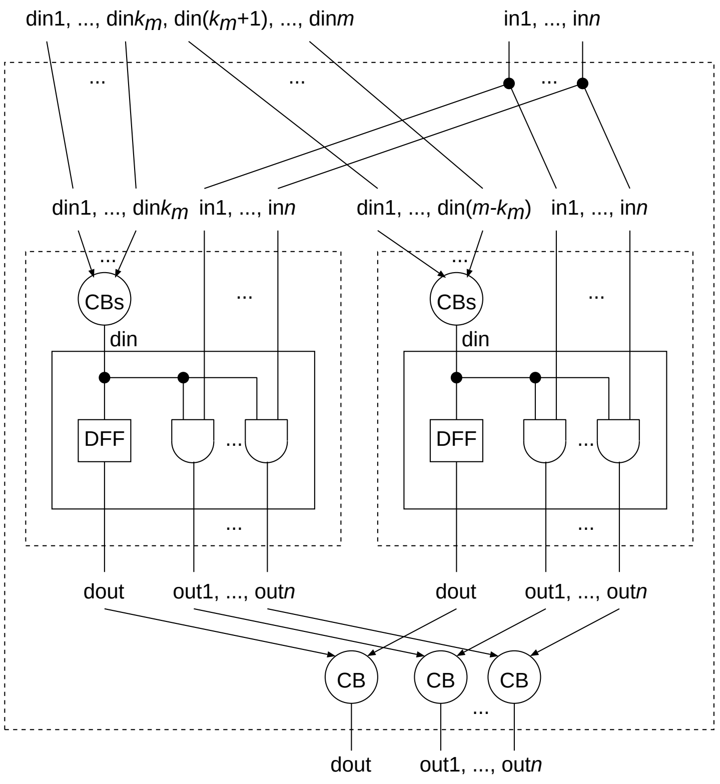


Figure 5.7: A case of $m = 2k_m$ and $n = k_n$.

used. Compared with the optimized cell designed for the state, the delay of SPLs for $din1$, . . ., $dinm$ connecting to din and the interconnections increases.

When $n > 2k_n$, we can also construct the state module using three or more basic cells in the same way.

A case of $m > k_m$ and $n > k_n$

By combining the methods of former two cases, we can construct the state module using basic cells.

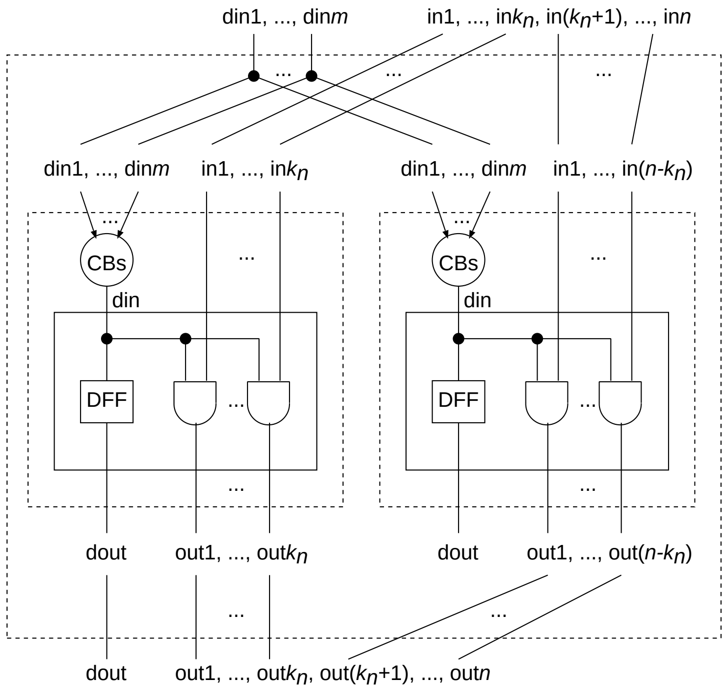


Figure 5.8: A case of $m = k_m$ and $n = 2k_n$.

Table 5.1: The delay of gates.

	delay (ps)
AND	20
OR	10
NOT	15
DFF	10
SPL	10
CB	20
PTL	10

5.4 Experimental Results

We have implemented the proposed method by programing language C and have applied it to synthesis of several sequential circuits in LGSynth'91 benchmark set[28]. We have also applied methods for CMOS digital circuits with binary encoding and one-hot encoding to the benchmark circuits. We have used SIS[33] and Synopsys Design Compiler for state minimization, state assignment and circuit synthesis. For timing adjustment, we have inserted buffers with clock input (DFFs) to appropriate positions of the sequential circuits. We have modified the method proposed in the previous chapter and have used for clock scheduling. We have estimated the number of gates and the clock periods.

We show the delay of gates which is used in the experiment in Table 5.1. For estimation of clock periods, we have assumed that gates are interconnected by PTLs[16–18]. Since the delay of PTLs is much smaller than that of gates, we have assumed that the delay of PTLs is a constant value regardless of their length and have approximated the delay by the delay of drivers and receivers of PTLs. Furthermore, we have assumed that all state modules and CBs connecting to their *din* are prepared as optimized cells (basic cells).

Tables 5.2 and 5.3 show the experimental results of synthesis of sequential circuits. In the tables, "# of inputs," "# of outputs" and "# of states" indicate the number of inputs, the number of outputs and the number of states after state minimization of each circuit, respectively. "proposed," "binary" and "one-hot" indicate the results of the proposed method, the method for CMOS digital circuits with binary encoding and that with one-hot encoding, respectively. "# of gates" indicates the number of gates (logic gates, DFFs and CBs) including buffers with clock input (DFFs) for timing adjustment. In the proposed method, each DFF and AND gate in state modules is counted as one gate. Since circuits synthesized by the proposed method work with a single clock signal, local clock frequency equals system clock frequency.

The experimental results show that compared with the conventional method for CMOS digital circuits with binary encoding, the proposed method synthesizes circuits that work with 4.9 times higher clock frequency and have 17.3% more gates on average. Compared with the conventional method for CMOS digital circuits with one-hot encoding, the proposed method synthesizes circuits that work with 3.1 times higher clock frequency and have 11.8% more gates on average.

5.5 Summary of the Chapter

For achieving high-throughput SFQ sequential circuits, a new synthesis method has been proposed. In the method, a state module consisting of a DFF and several AND gates is used. First, the states of a sequential machine are encoded by one-hot encoding, and then, a state module is assigned to each state. Finally, according to the state transition, the state modules are connected with each other using CBs. Using the proposed method, a sequential circuit without clocked gates in its feedback loops is constructed. The experimental results show that compared with a conventional method for CMOS digital circuits, the proposed method synthesizes circuits that work with 4.9 times higher clock frequency and have 17.3% more gates on average.

In general, the number of gates in CMOS sequential circuits with one-hot encoding is larger than that with binary encoding. However, in an SFQ sequential circuit, buffers with clock input (DFFs) for timing adjustment are necessary, and therefore, the number of gates in sequential circuits with one-hot encoding is comparable with that in sequential circuits with binary encoding. The proposed method is a method applicable for arbitrary types of sequential machines and high-throughput circuits are synthesized with comparable number of gates with conventional methods.

Table 5.2: Experimental results of sequential circuit synthesis (1).

Circuit name[28]	# of inputs	# of outputs	# of states		# of gates	Local clock (GHz)	System clock (GHz)
dk15	3	5	4	proposed	206	8.70	8.70
				binary	105	15.38	2.20
				one-hot	132	13.33	2.22
dk16	2	3	27	proposed	299	7.41	7.41
				binary	402	13.33	1.21
				one-hot	300	18.18	2.60
ex1	9	19	18	proposed	547	6.90	6.90
				binary	397	13.33	1.33
				one-hot	376	15.38	2.20
ex2	2	2	11	proposed	132	8.00	8.00
				binary	131	13.33	1.67
				one-hot	110	18.18	3.03
ex5	2	2	4	proposed	56	8.00	8.00
				binary	33	18.18	3.03
				one-hot	47	18.18	3.64
keyb	7	2	19	proposed	428	5.71	5.71
				binary	527	15.38	1.18
				one-hot	383	18.18	1.82
kirkman	12	6	17	proposed	515	6.06	6.06
				binary	347	13.33	1.33
				one-hot	519	16.67	2.08
lion9	2	1	4	proposed	36	9.52	9.52
				binary	33	18.18	3.03
				one-hot	38	18.18	3.64
planet	7	19	48	proposed	613	9.52	9.52
				binary	930	12.50	1.14
				one-hot	706	18.18	2.60
pma	8	8	24	proposed	435	8.70	8.70
				binary	385	13.33	1.33
				one-hot	375	18.18	2.60

Table 5.3: Experimental results of sequential circuit synthesis (2).

Circuit name[28]	# of inputs	# of outputs	# of states		# of gates	Local clock (GHz)	System clock (GHz)
s1	8	6	20	proposed	566	6.90	6.90
				binary	655	13.33	1.11
				one-hot	399	13.33	1.67
s1488	8	19	48	proposed	741	5.41	5.41
				binary	1003	13.33	1.11
				one-hot	889	15.38	1.71
s27	4	1	5	proposed	109	8.00	8.00
				binary	38	18.18	3.03
				one-hot	103	18.18	2.27
s298	3	6	135	proposed	1472	4.65	4.65
				binary	2749	13.33	0.83
				one-hot	1718	18.18	2.02
s386	7	7	13	proposed	270	6.45	6.45
				binary	227	15.38	1.54
				one-hot	187	18.18	2.60
s820	18	19	24	proposed	648	5.71	5.71
				binary	535	13.33	1.11
				one-hot	609	18.18	1.82
sand	11	9	32	proposed	873	6.90	6.90
				binary	924	11.76	1.07
				one-hot	750	15.38	1.71
scf	27	56	94	proposed	927	5.13	5.13
				binary	1522	13.33	1.03
				one-hot	1460	15.38	1.54
shiftreg	1	1	8	proposed	35	10.53	10.53
				binary	20	25.64	6.41
				one-hot	25	18.18	6.06
styr	9	10	30	proposed	668	5.71	5.71
				binary	850	13.33	1.03
				one-hot	664	18.18	2.02

Chapter 6

Integer Multiplier with Systolic Array Structure

6.1 Introduction

In this chapter, we propose an integer multiplier for concurrent-flow clocking SFQ digital circuits based on the systolic array scheme. Since SFQ digital circuits work by pulse logic, logic gates of SFQ digital circuits have different features from those of CMOS digital circuits. For example, all logic gates of synchronous clocking SFQ digital circuits have the function of a latch and NOT gate of dual-rail SFQ digital circuits has no cost, i.e., cross of *true*-line and *false*-line. Therefore, suitable circuit structure of SFQ arithmetic circuits is different from that of CMOS arithmetic circuits. The importance of studies on such suitable circuit structure is increasing. In this chapter, we focus on multiplication because multiplication appears frequently in various applications. Since multiplication using addition and shift operations requires relatively long calculation time, multipliers are integrated into many commercial microprocessors.

The systolic array is a parallel architecture and is suitable for VLSI implementation[34]. It consists of regularly arranged simple processing elements (PEs). The systolic integer multiplier to be proposed has one-dimensional structure, and all signals of the multiplier flow from the circuit input to the output unidirectionally. The multiplier matches concurrent-flow clocking of SFQ digital circuits well. The circuit area of the proposed multiplier is proportional to the bit-width of the operand and the latency required for a multiplication is also proportional to the bit-width of the operand.

For evaluation of the proposed multiplier, we have designed a 4-bit systolic multiplier. For comparison, we have also designed a 4-bit array multiplier which is one of the most typical parallel multipliers. From the results of the design and a digital simulation, the number of JJs and the latency of the designed 4-bit systolic multiplier are 2308 and 1240 ps at 25 GHz of clock frequency, respectively, while those of the designed 4-bit array multiplier are 4254 and 840 ps, respectively. We have estimated the performance of larger scale multipliers. The estimated latency of n-bit systolic multiplier is $(322.5n - 40)$ ps at 25 GHz of clock frequency, while that of n-bit array multiplier is $(252n - 168)$ ps. The ratio is about 1.28 when n is large. The circuit area of the proposed multiplier is proportional to n, while that of the array multiplier is proportional to n^2. Our estimation shows that the systolic multiplier achieves comparable latency to the array multiplier, using extremely fewer number of JJs when the bit-width of the operand is large. We have fabricated a 1-bit PE of the proposed multiplier using NEC 2.5kA/cm^2 standard Nb process and have successfully tested it at low speed.

In general, for CMOS digital circuits, latency of an arithmetic circuit with systolic array structure is long because of the setup/hold time of registers included in the systolic array. For synchronous clocking SFQ digital circuits, all logic gates have the function of a latch. Therefore, additional delay is not required in the systolic array, and latency of an SFQ arithmetic circuit with systolic array structure is not long in comparison with typical architectures. Adoption of systolic array structure is a promising approach for designing SFQ arithmetic circuits.

The rest of this chapter is organized as follows. In Section 6.2, we describe the characteristics of the systolic array and propose an integer multiplier with systolic array structure. In Section 6.3, we describe the results of design and evaluation of the proposed multiplier and in Section 6.4, we describe the test results. Finally, in Section 6.5, we give the summary of this chapter.

6.2 Integer Multiplier with Systolic Array Structure

6.2.1 Systolic Array

The systolic array was proposed as an architecture suitable for VLSI implementation[34]. We review the characteristics of a systolic array briefly.

A systolic array consists of a set of interconnected processing elements (systolic cells) and each cell performs some simple operation. Systolic cells are interconnected to form simple, regular communication and control structure. Signals in a systolic array flows between cells in a pipelined fashion, and communication with the outside world occurs only at the boundary cells. A systolic array has the following advantages:

- Modular expandability,
- Simple and regular data and control flows,
- Use of simple and uniform cells,
- No global broadcasting.

Each systolic cell has finite number of registers. All signals except the clock signal of a cell are only connected to its adjacent cells. Data signals are input serially and are output serially. The systolic array achieves high-throughput operations and small circuit area, while latency tends to be long.

6.2.2 Integer Multiplier with Systolic Array Structure

We consider the multiplication $Z = X \times Y$ where X is a multiplicand, Y is a multiplier and Z is the product. X and Y are n-bit two's complement integers and Z is a $2n$-bit two's complement integer. In this section, first, we transform the expression of multiplication for constructing a systolic array. Then, we propose an integer multiplier from the transformed expression.

X and Y can be written as follows:

$$X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i, \tag{6.1}$$

$$Y = -y_{n-1}2^{n-1} + \sum_{j=0}^{n-2} y_j 2^j. \tag{6.2}$$

Then, the expression is rewritten as follows[35]:

$$\begin{aligned} Z &= X \times Y \\ &= \left(-x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i\right) \times \left(-y_{n-1}2^{n-1} + \sum_{j=0}^{n-2} y_j 2^j\right) \\ &= x_{n-1}y_{n-1}2^{2n-2} - x_{n-1}\sum_{j=0}^{n-2} y_j 2^{n+j-1} \\ &\quad -y_{n-1}\sum_{i=0}^{n-2} x_i 2^{n+i-1} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} x_i y_j 2^{i+j}. \end{aligned} \tag{6.3}$$

To remove subtractions from Eq. (6.3), we use the following relations:

$$\begin{aligned} -x_{n-1}\sum_{j=0}^{n-2} y_j 2^{n+j-1} &= \left(-2^{n-1} + \sum_{j=0}^{n-2} \overline{x_{n-1}y_j}2^j + 1\right)2^{n-1}, \\ -y_{n-1}\sum_{i=0}^{n-2} x_i 2^{n+i-1} &= \left(-2^{n-1} + \sum_{i=0}^{n-2} \overline{y_{n-1}x_i}2^i + 1\right)2^{n-1}. \end{aligned}$$

$\overline{a}$ is the bit-complement of a. Then, Eq. (6.3) can be rewritten as follows:

$$\begin{aligned} Z &= -2^{2n-1} + x_{n-1}y_{n-1}2^{2n-2} + \sum_{j=0}^{n-2} \overline{x_{n-1}y_j}2^{n+j-1} \\ &\quad + \sum_{i=0}^{n-2} \overline{y_{n-1}x_i}2^{n+i-1} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} x_i y_j 2^{i+j} + 2^n. \end{aligned} \tag{6.4}$$

Based on Eq. (6.4), we propose a systolic integer multiplier. The multiplier to be proposed has one-dimensional structure as shown in Fig. 6.1. In the figure, a 4-bit multiplier is shown. x_i (y_j) is input 1-bit per clock cycle from the LSB (least significant bit), and $z_0, \ldots, z_{2n-1}$ are serially obtained at the right-most cell. In cell j, y_j is stored and $y_j \cdot x_i$

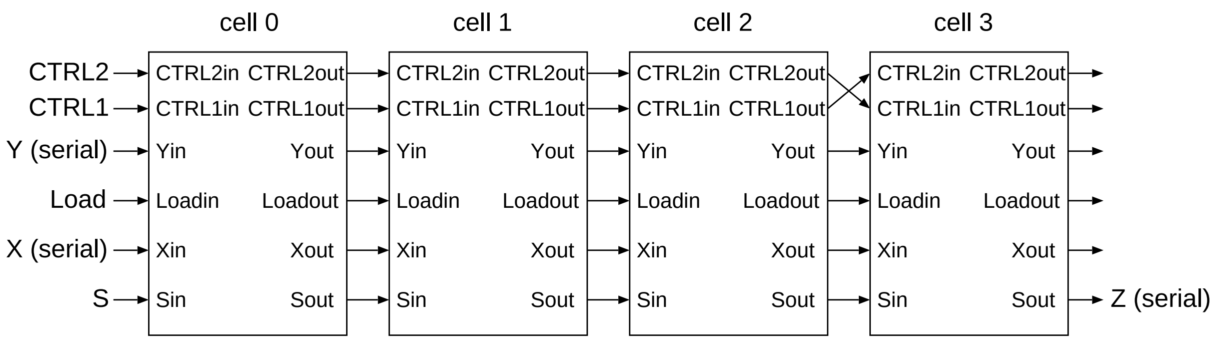


Figure 6.1: 4-bit integer multiplier.

$(i = 0, \ldots, n-1)$ are calculated. The calculated $y_j \cdot x_i$ is accumulated with an intermediate result which is input from *Sin*, and the result of the accumulation is output from *Sout* to the next cell. *Load* is used for loading Y and resetting each cell. y_j is loaded to an appropriate cell according to the load signal which is fed to *Load* at the same time of the input of y_0. *CTRL1* and *CTRL2* are used for the bit-complementation included in Eq. (6.4) and are exchanged each other at the right-most cell.

The detail of the 1-bit cell is shown in Fig. 6.2. The cell consists of 2 non-destructive read-outs (NDs), 2 ANDs, 3 XORs, 1 confluence buffer (CB), 8 splitters and 21 DFFs (Ds), and is a serial adder that calculates $(X \cdot Y) \oplus \mathit{CTRL1} + S$. The NDs calculate $X \cdot Y$ and control loading Y. Signals input from *Sin* and *Yin* are output at *Sout* and *Yout* after 3 clock cycles, respectively, and signals input from *Xin*, *Loadin*, *CTRL1in* and *CTRL2in* are output at *Xout*, *Loadout*, *CTRL1out* and *CTRL2out* after 4 clock cycles, respectively. After $3n$ clock cycles from the input of x_0 to the left-most cell, z_0 is obtained from the right-most cell and after then, $z_1, \ldots, z_{2n-1}$ are serially obtained.

For the bit-complementation in Eq. (6.4), we set $\mathit{CTRL1} = 1$ when x_{n-1} is input from X and $\mathit{CTRL2} = 1$ when $x_0, \ldots, x_{n-2}$ are input. Furthermore, for the addition of 2^n, we input 1 from S at the $(n+1)$-th clock cycle and for the addition of -2^{2n-1}, we input 1 from S at the $2n$-th clock cycle.

We show the flow of a 4-bit multiplication in Fig. 6.3. The multiplication starts at

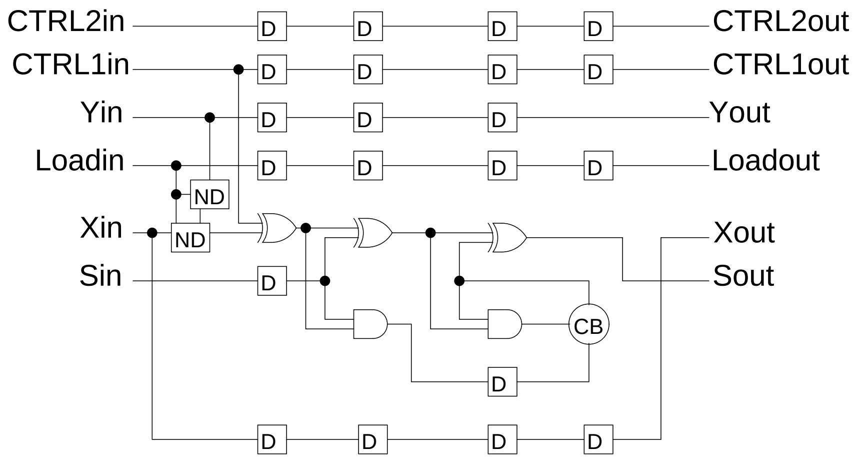


Figure 6.2: 1-bit cell of the proposed multiplier. (D is a DFF, ND is a non-destructive read-out (NDRO) and CB is a confluence buffer.)

$t = 1$ and ends at $t = 20$. The product Z is obtained during $t = 13$ to $t = 20$. The next multiplication can start at $t = 9$. Each box separated by dashed lines indicates a pipeline stage of the 1-bit cell, each box filled by oblique lines indicates a pipeline stage which is working and each dot indicates an intermediate step of an addition.

All signals of the proposed integer multiplier flow from the circuit input to the output unidirectionally. This feature matches concurrent-flow clocking of SFQ digital circuits well. The circuit area of the proposed multiplier is proportional to n which is the bit-width of the operand and the latency of the multiplier is also proportional to n.

6.3 Design and Evaluation of the Proposed Multiplier

In order to evaluate the proposed systolic integer multiplier, we have designed a 4-bit systolic integer multiplier. For comparison, we have also designed a 4-bit array integer multiplier which is one of the most typical parallel multipliers. We have used CONNECT cells[5] and have assumed NEC 2.5kA/cm^2 standard Nb process[4]. Figure 6.4 shows the circuit structure of a 4-bit array integer multiplier. "FA" indicates a full adder. For

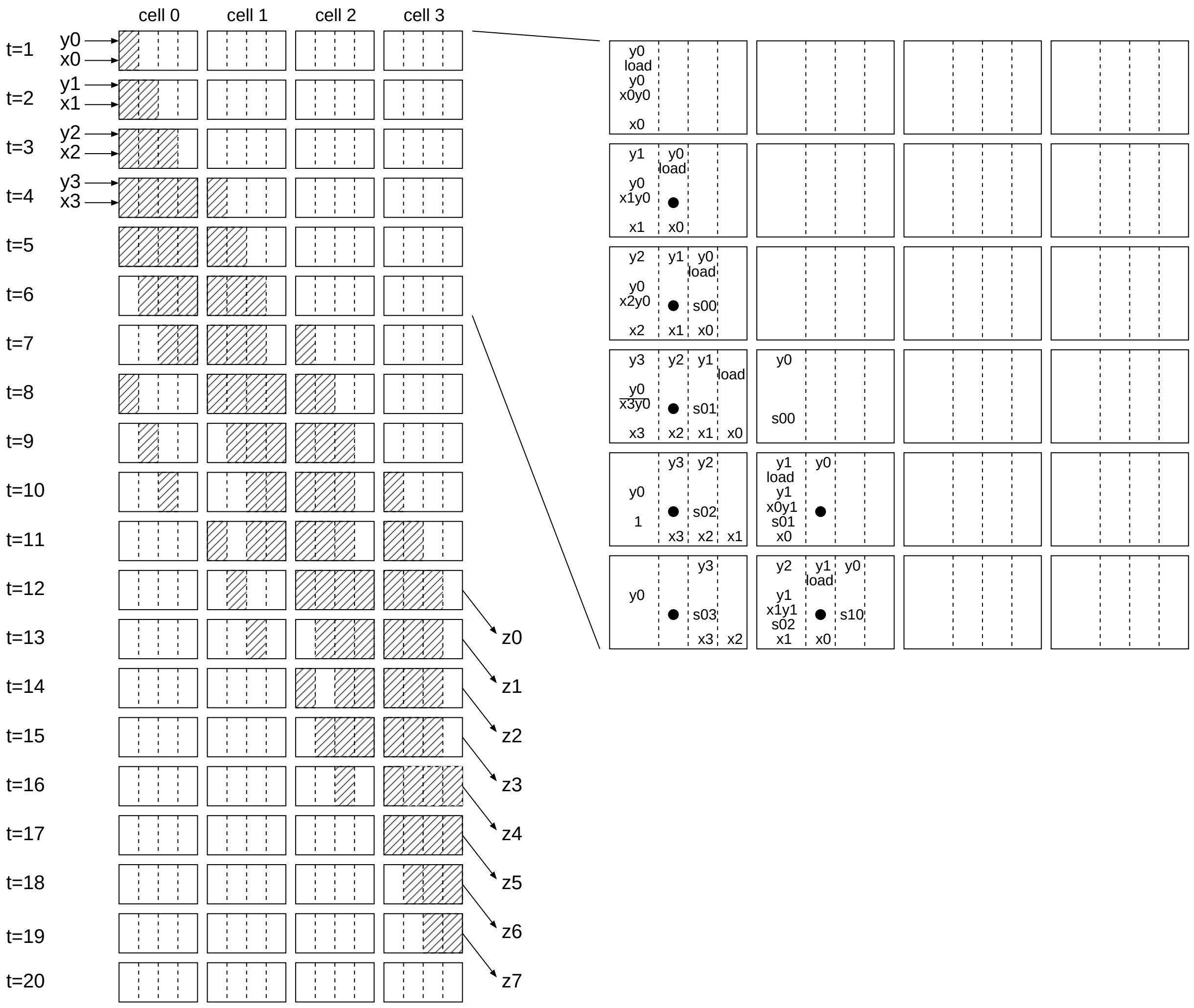


Figure 6.3: Flow of a 4-bit multiplication.

interconnections of the systolic multiplier, we have used only Josephson-transmission-lines (JTLs) because all interconnections of the multiplier are short. On the other hand, for interconnections of the array multiplier which has many long interconnection, we have assumed that passive-transmission-lines (PTLs) are available without any restrictions and the delay of PTLs is 0 (We have taken the delay of drivers and receivers of PTLs into consideration).

We have compared the 4-bit multipliers with each other. Table 6.1 shows the number

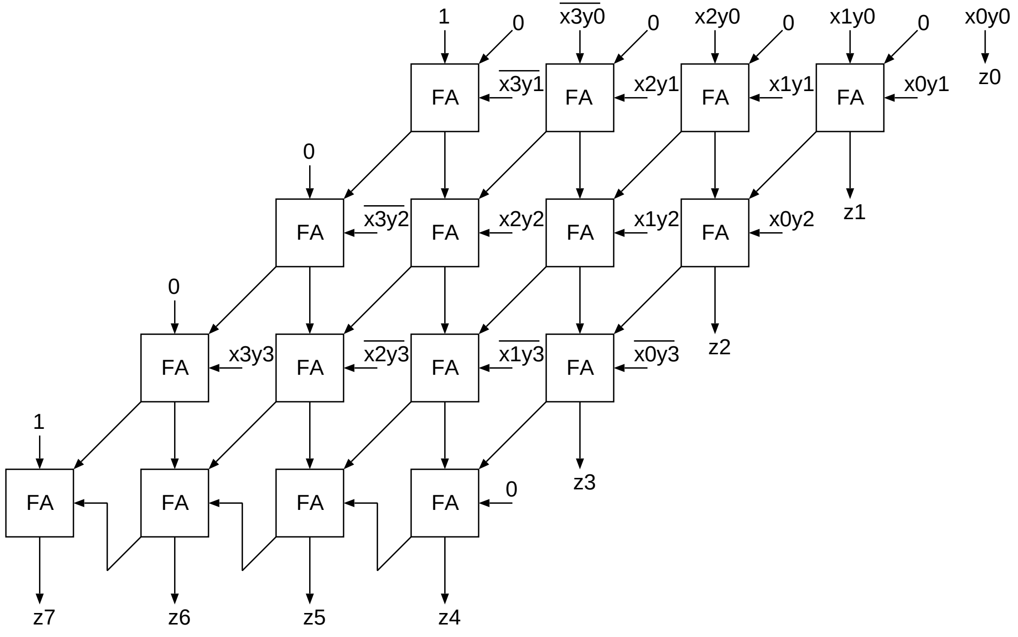


Figure 6.4: Circuit structure of a 4-bit array integer multiplier.

of JJs and the latency of the designed multipliers. The clock frequency of the multipliers is 25 GHz (The clock period is 40 ps). The propagation delay of the clock signal is its delay from the input to the output. The number of stages is the number of gates with clock input on the paths from the circuit input to the output. Latency is approximated by the following expression: *latency* = *((the number of clocks required* -1*)* $\times$ *the clock period)* + *the propagation delay of the clock signal*. The number of JJs of the systolic multiplier is almost the half of that of the array multiplier. The latency of the former is about 1.5 times longer than that of the latter.

Here, we estimate the performance of larger scale multipliers. Since an n-bit systolic multiplier can be constructed by connecting n 1-bit cells in serial, the number of JJs, the propagation delay of the clock signal and the number of required clocks for an n-bit systolic multiplier are all proportional to n. Therefore, the latency of n-bit systolic multiplier is approximated by $(5n-1)\times 40+490/4\times n=(322.5n-40)$ ps. Since circuit area of an n-bit array multiplier is proportional to n^2, we assume that the number of

Table 6.1: The number of JJs and the latency of the designed multiplier.

	4-bit systolic multiplier	4-bit array multiplier
# of JJs	2308	4254
propagation delay of the clock signal	490	440
# of pipeline stages	12	10
# of clocks required	20	11
The latency at 25 GHz (ps)	1250	840

JJs of an n-bit array multiplier is exactly proportional to n^2. For estimating latency of an array multiplier, we assume that propagation delay of the clock signal is exactly proportional to the number of stages of logic gates with clock input. The number of stages of n-bit array multiplier is $(3n - 2)$. Namely, those of an 8, 16, 32 and 64-bit multipliers are 22, 46, 94 and 190, respectively. Therefore, the numbers of required clocks of the multipliers are 23, 47, 95 and 191, respectively. We assume clock frequency of all multipliers is 25 GHz. Consequently, the latency of n-bit array multiplier is approximated by $(3n - 1) \times 40 + 440/10 \times (3n - 2) = (252n - 168)$ ps. Tables 6.2 and 6.3 show the estimated numbers of JJs and the estimated latency, respectively. The latency of the 64-bit systolic multiplier is about 1.3 times longer than that of the 64-bit array multiplier. Since the delay of PTLs is not taken into consideration, the ratio is small if it is taken. The proposed multiplier achieves comparable latency to the array multiplier, using extremely fewer number of JJs when the bit-width of the operand is large.

In general, for CMOS digital circuits, latency of an arithmetic circuit with systolic array structure is long in comparison with typical architectures because of the setup/hold time of registers included in systolic cells. On the other hand for SFQ digital circuits, latency of an arithmetic circuit with systolic array structure is not long in comparison

Table 6.2: The estimated number of JJs of larger scale multipliers.

	systolic	array	ratio (systolic / array)
4-bit	2308	4254	0.5425
8-bit	4616	17016	0.2713
16-bit	9232	68064	0.1356
32-bit	18464	272256	0.0678
64-bit	36928	1089024	0.0339

Table 6.3: The estimated latency of larger scale multipliers.

	systolic (ps)	array (ps)	ratio (systolic / array)
4-bit	1250	840	1.49
8-bit	2540	1848	1.37
16-bit	5120	3864	1.33
32-bit	10280	7896	1.30
64-bit	20600	15960	1.29

with typical architectures because of the function of a latch all logic gates have. Systolic array structure is suitable for SFQ arithmetic circuits.

The proposed multiplier can start a multiplication every $2n$ clock cycles, while the array multiplier can start a multiplication every 1 clock cycle. The proposed multiplier is attractive for the situation where multiplication is not so frequently appeared.

6.4 Test Result of the Proposed Multiplier

We have fabricated a 1-bit cell of the proposed systolic integer multiplier using NEC 2.5kA/cm^2 standard Nb process. Figure 6.5 shows a chip photograph of the 1-bit cell. The number of JJs of the cell is 577 and the circuit area of it is 0.60 mm $\times$ 0.56 mm. Figure 6.6 shows the test result of the 1-bit cell at low speed. We have set $Y = 1$ and have tested the cell at two situations, i.e., $CTRL1 = 0$ (the left part) and $CTRL1 = 1$ (the right

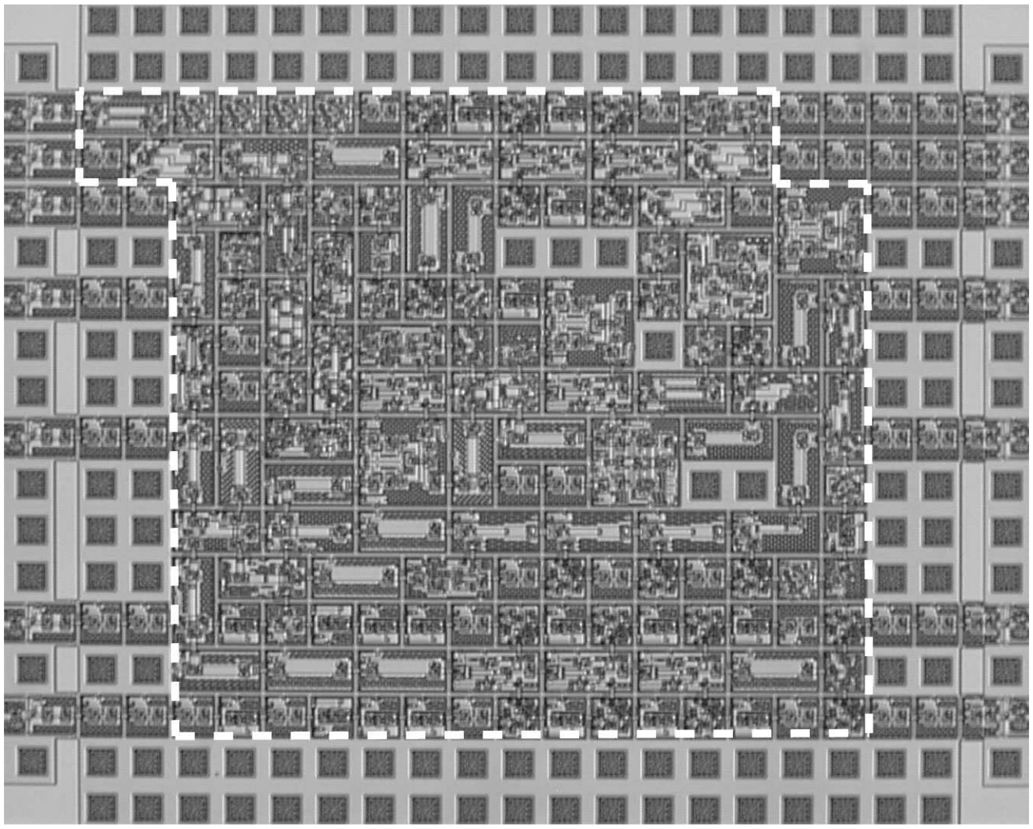


Figure 6.5: Chip photograph of the 1-bit cell. (The number of JJs is 577 and the circuit area is 0.60 mm × 0.56 mm.)

part). As test patterns, we have input 0101 from *Sin* and 0110 from *Xin*, respectively. The expected results of the two situations are 1011 and 1110, respectively because the cell is a serial adder that calculates $(X \cdot Y) \oplus \mathit{CTRL1} + S$. We can see that the cell works correctly. The dc bias margin of the cell ranges from -5% to 11%.

6.5 Summary of the Chapter

We have proposed a systolic integer multiplier and have designed a 4-bit version using CONNECT cells. The number of JJs of the 4-bit systolic multiplier is almost the half of a 4-bit array multiplier, and its latency is about 1.5 times longer than that of the

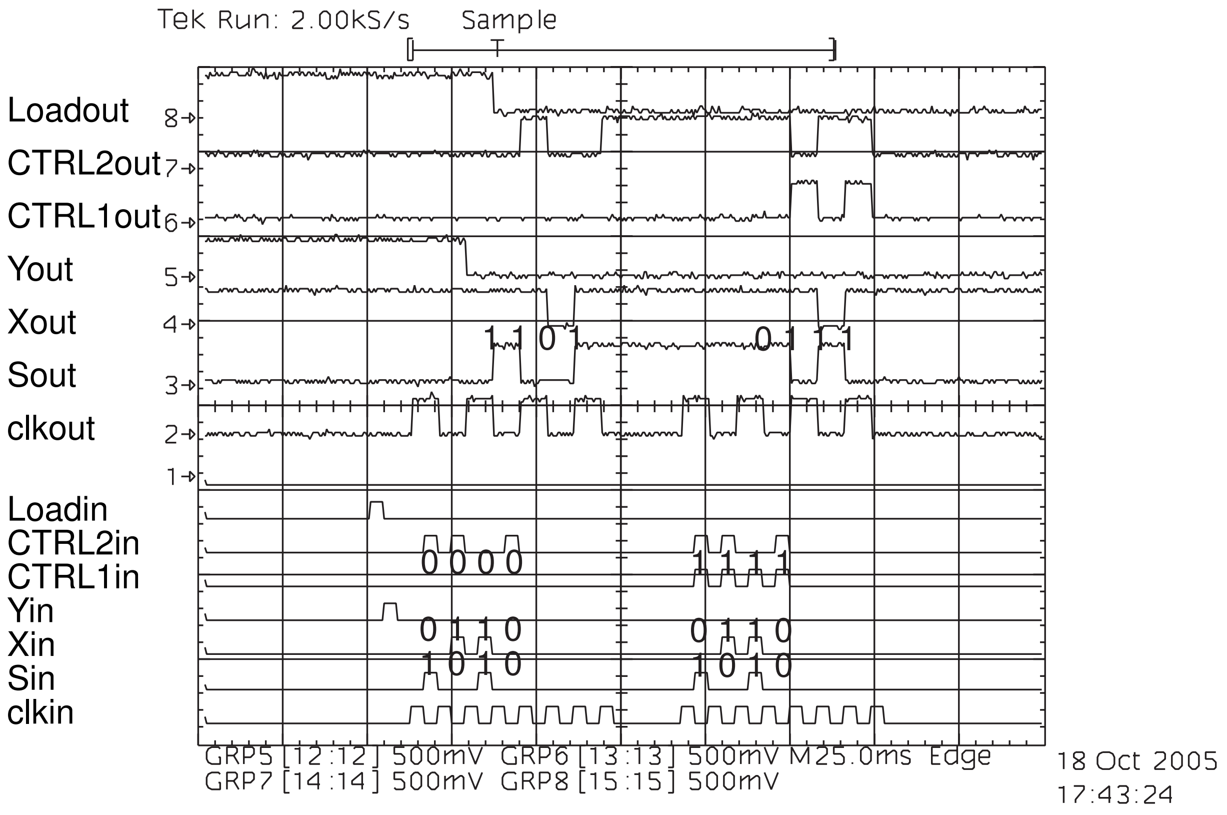


Figure 6.6: Test result of the 1-bit cell. (The test pattern of the left side is $Y = 1$, $CTRL1 = 0$, $S = 0101$ and $X = 0110$, and the result is 1011. The test pattern of the right side is $Y = 1$, $CTRL1 = 1$, $S = 0101$ and $X = 0110$, and the result is 1110.)

array multiplier. Our estimation of the performance of larger scale multipliers shows that the proposed systolic multiplier achieves comparable latency to the array multiplier, using extremely fewer number of JJs when the bit-width of the operand is large. We have fabricated a 1-bit cell of the proposed multiplier using NEC 2.5kA/cm^2 standard Nb process and have successfully tested it at low speed. The proposed systolic multiplier fully exploits the ultra-fast computation speed of SFQ circuits. By using systolic array structure, a small-area and fast multiplier is achieved.

In general, for CMOS digital circuits, latency of an arithmetic circuit with systolic array structure is long in comparison with typical architectures because of the setup/hold time of registers included in systolic cells. On the other hand for SFQ digital circuits, latency of an arithmetic circuit with systolic array structure is not long in comparison with typical architectures because of the function of a latch all logic gates have. Adoption of systolic array structure is a promising approach for designing high-speed and small-area SFQ arithmetic circuits.

Chapter 7

Conclusion

We proposed design automation algorithms and design of an arithmetic circuit for SFQ digital circuits.

In Chapter 3, we proposed a new method of logic synthesis for dual-rail SFQ digital circuits. In the method, we constructed a root-shared binary decision diagram (RSBDD) from given logic functions and reduced the size of the constructed RSBDD by a variable re-ordering technique. Then, we constructed a dual-rail SFQ circuit using the reduced RSBDD. The experimental results showed that by using the proposed method, we can synthesize circuits that consist of about 27.1% fewer logic elements than those synthesized by a Transduction-based method on average.

In Chapter 4, we proposed an algorithm for clock scheduling of concurrent-flow clocking SFQ digital circuits. By using the algorithm, we can determine clock scheduling for a given clock period in the computation time of $O(g^3)$ where g is the number of clocked gates. Experimental results on smaller benchmark circuits showed that the proposed algorithm can obtain the optimum solutions, i.e., the same results with an ILP solver. Experimental results on larger benchmark circuits showed that for a given clock period, the proposed algorithm can obtain near optimal solutions in which inserted delay elements are 59.0% fewer and clock trees are 40.4% shorter on average than those by a straightforward algorithm. The proposed algorithm can also be used to minimize the clock period. The clock periods obtained by the proposed algorithm were 19.0% shorter on average than those by the straightforward algorithm.

In Chapter 5, we proposed a new synthesis method of sequential circuits for synchronous clocking SFQ digital circuits. In the method, we used a state module consisting of a DFF and several AND gates. First, we encoded the states of a sequential machine by one-hot encoding, and then, assigned a state module to each state. Finally, we connected the state modules with each other according to the state transition using CBs. Using the proposed method, we can construct a sequential circuit without clocked gates in its feedback loops. The experimental results showed that compared with a conventional method for CMOS digital circuits, the proposed method can synthesize circuits that work with 4.9 times higher clock frequency and have 17.3% more gates on average.

In Chapter 6, we proposed a systolic integer multiplier for concurrent-flow clocking SFQ digital circuits and designed a 4-bit version using CONNECT cells. The number of JJs of the 4-bit systolic multiplier was almost the half of a 4-bit array multiplier, and its latency was about 1.5 times longer than that of the array multiplier. Our estimation of the performance of larger scale multipliers showed that the proposed systolic multiplier can achieve comparable latency to the array multiplier, using extremely fewer number of JJs when the bit-width of the operand is large. We fabricated a 1-bit cell of the proposed multiplier using NEC 2.5kA/cm^2 standard Nb process and successfully tested it at low speed.

In this dissertation, we focused on logic synthesis and clock scheduling from various topics of studies of CAD systems. To use the proposed design automation algorithms effectively, development of placement and routing tools suitable for SFQ digital circuits is important. When we designed the algorithms, we assumed that the delay of PTLs is ignorable. We need to deal with the delays in the placement and routing phase. More precisely, we need to minimize or equalize the length of each PTL in the placement and routing phase. In addition, evaluation of errors caused by process variation, timing jitter and so on is also important because the errors affect the robustness of designed circuits.

These topics are especially important in designing larger scale SFQ digital circuits.

Through the studies in this dissertation, much knowledge about CAD systems for SFQ digital circuits was obtained. By the method of logic synthesis proposed in Chapter 3, small-area dual-rail circuits were achieved. The result shows that adoption of dual-rail representation has the possibilities to be an alternative approach for designing SFQ digital circuits. By the clock scheduling algorithm proposed in Chapter 4, high-throughput circuits were achieved with fewer delay elements. The result shows that appropriate clock scheduling makes the performance of circuits higher and the proposed algorithm is an effective for the clock scheduling problem. By the method of sequential circuit synthesis proposed in Chapter 5, sequential circuits which work with several times higher clock frequency were achieved. High-throughput sequential circuits can be designed systematically using the proposed method. Utilization of SFQ-specific gates and development of new design methods make the performance of SFQ digital circuits higher. The obtained knowledge will be bases of the development of CAD systems for SFQ digital circuits. In the study of the integer multiplier, a systolic multiplier achieved comparable latency to an array multiplier with extremely smaller circuit area. This result is valuable knowledge to design SFQ arithmetic circuits. Adoption of systolic array structure is a promising approach to design high-speed arithmetic circuits with smaller circuit area. Indeed, some arithmetic circuits have been designed based on the systolic array scheme[36, 37]. Advancement of CAD systems and process technology will lead SFQ circuit technology to a major player in future information and communication technology.

Bibliography

[1] K.K. Likharev and V.K. Semenov. RSFQ logic/memory family: A new josephson-junction technology for sub-terahertz-clock-frequency digital systems. *IEEE Trans. Appl. Supercond.*, 1(1):3–28, March 1991.

[2] International Technology Roadmap for Semiconductor. *Emerging research devices 2003 edition*, http://public.itrs.net/, 2003.

[3] K. Gaj, E.G. Friedman, and M.J. Feldman. Timing of multi-gigahertz rapid single flux quantum digital circuits. *Journal of VLSI Signal Processing*, 16(2/3):247–276, June/July 1997.

[4] S. Nagasawa, Y. Hashimoto, H. Numata, and S. Tahara. A 380 ps, 9.5 mW Josephson 4-Kbit RAM operated at a high bit yield. *IEEE Trans. Appl. Supercond.*, 5(2):2447–2452, June 1995.

[5] S. Yorozu, Y. Kameda, H. Terai, A. Fujimaki, T. Yamada, and S. Tahara. A single flux quantum standard logic cell library. *Physica C*, 378–381:1471–1474, October 2002.

[6] N. Yoshikawa, T. Nishigai, H. Kojima, K. Fujiwara, A. Fujimaki, T. Yamada, M. Tanaka, S. Yorozu, M. Hidaka, and H. Terai. Magnetic shielding against DC bias current toward large-scale SFQ integrated circuits. In *Appl. Supercond. Conf.*, Jacksonville, Florida, October 2004.

[7] M. Dorojevets, P. Bunyk, and D. Zinoviev. FLUX Chip: Design of a 20-GHz 16-bit ultrapipelined RSFQ processor prototype based on 1.75-μm LTS technology. *IEEE Trans. Appl. Supercond.*, 11(1):326–332, March 2001.

[8] A. Fujimaki, Y. Takai, and N. Yoshikawa. High-end server based on complexity-reduced architecture for superconductor technology. *IEICE Trans. Electron.*, E85-C(3):612–616, March 2002.

[9] M. Tanaka, F. Matsuzaki, T. Kondo, N. Nakajima, Y. Yamanashi, A. Fujimaki, H. Hayakawa, N. Yoshikawa, H. Terai, and S. Yorozu. A single-flux-quantum logic prototype microprocessor. In *Technical Digest of IEEE International Solid-State Circuits Conference (ISSCC) 2004*, pages 298–299, San Francisco, USA, February 2004.

[10] M. Tanaka, T. Kondo, N. Nakajima, T. Kawamoto, Y. Yamanashi, Y. Kamiya, A. Akimoto, A. Fujimaki, H. Hayakawa, N. Yoshikawa, H. Terai, Y. Hashimoto, and S. Yorozu. Demonstration of a single-flux-quantum microprocessor using passive transmission lines. *IEEE Trans. Appl. Supercond.*, 15(2):400–404, June 2005.

[11] M. Tanaka, T. Kawamoto, Y. Yamanashi, Y. Kamiya, A. Akimoto, K. Fujiwara, A. Fujimaki, N. Yoshikawa, H. Terai, and S. Yorozu. Design of a pipelined 8-bit-serial single-flux-quantum microprocessor with multiple ALUs. *Supercond. Sci. Technol.*, 19(5):344–349, March 2006.

[12] N. Takagi, K. Murakami, A. Fujimaki, N. Yoshikawa, K. Inoue, and H. Honda. A processor with a large-scale reconfigurable data-path using rapid single flux quantum circuits. In *IEICE Tech. Rep.*, SCE2006-36, pages 37–40, January 2007. (in Japanese).

[13] Y. Kameda, S. Yorozu, M. Hidaka, and S. Tahara. Successful operation of single-flux-quantum 2x2 unit switch. *Physica C*, 378–381:1466–1470, October 2002.

[14] M. Tanaka, Y. Yamanashi, Y. Kamiya, A. Akimoto, N. Irie, Hee-Joung Park, A. Fujimaki, N. Yoshikawa, H. Terai, and S. Yorozu. A new design approach for high-throughput arithmetic circuits for single-flux-quantum microprocessors. *IEEE Trans. Appl. Supercond.*, 17(2):516–519, June 2007.

[15] Hee-Joung Park, Y. Yamanashi, N. Yoshikawa, M. Tanaka, A. Fujimaki, H. Terai, and S. Yorozu. Design and implementation of the 4-b bit-slice adder using SFQ circuits. In *IEICE Tech. Rep.*, SCE2006-32, pages 13–18, January 2007. (in Japanese).

[16] S.V. Polonsky, V.K. Semenov, and D.F. Schneider. Transmission of single-flux-quantum pulses along superconducting microstrip lines. *IEEE Trans. Appl. Supercond.*, 3(1):2598–2600, March 1993.

[17] T. Yamada, H. Ryoki, A. Fujimaki, and S. Yorozu. Flexible superconducting passive interconnects with 50-Gb/s signal transmissions in single-flux-quantum circuits. *Jpn. J. Appl. Phys.*, 45(2A):752–757, 2006.

[18] Y. Hashimoto, S. Yorozu, T. Satoh, and T. Miyazaki. Demonstration of chip-to-chip transmission of single-flux-quantum pulses at throughputs beyond 100Gbps. *Appl. Phys. Lett.*, 87(2):022502, 2005.

[19] Y. Hashimoto, S. Yorozu, T. Miyazaki, Y. Kameda, H. Suzuki, and N. Yoshikawa. Implementation and experimental evaluation of a cryocooled system prototype for high-throughput SFQ digital applications. *IEEE Trans. Appl. Supercond.*, 17(2):546–551, June 2007.

[20] N. Yoshikawa and J. Koshiyama. Top-down RSFQ logic design based on a binary decision diagram. *IEEE Trans. Appl. Supercond.*, 11(1):1098–1101, March 2001.

[21] P. Patra, S. Polonsky, and D.S. Fussell. Delay insensitive logic for RSFQ superconductor technology. In *Proc. 3rd International Symposium on Advanced Research*

in Asynchronous Circuits and Systems (ASYNC '97), pages 42–53, Eindhoven, The Netherlands, April 1997.

[22] Y. Kameda, S. Polonsky, M. Maezawa, and T. Nanya. Primitive-level pipelining method on delay-insensitive model for RSFQ pulse-driven logic. In *Proc. 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 262–273, San Diego, CA, March 1998.

[23] K. Tateishi, Y. Kameda, and S. Yorozu. High-speed datapath P&R technique for cell-based single flux quantum circuit. *IEICE Trans. Fundamentals*, J88-A(3):330–337, March 2005. (in Japanese).

[24] S. Yamashita, K. Tanaka, H. Takada, K. Obata, and K. Takagi. A transduction-based framework to synthesize RSFQ circuits. In *Proc. ASP-DAC 2006*, pages 266–272, Yokohama, Japan, 2006.

[25] S.B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, C-27(6):509–516, June 1978.

[26] R.E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, August 1986.

[27] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. ICCAD'93*, pages 42–47, Santa Clara, CA, 1993.

[28] S. Yang. Logic synthesis and optimization benchmarks user guide ver.3.0. Technical report, Microelectronics Center of North Carolina, January 1991.

[29] Y. Kohira and A. Takahashi. Clock period minimization method of semi-synchronous circuits by delay insertion. *IEICE Trans. Fundamentals*, E88-A(4):892–898, April 2005.

[30] S-H. Huang and Y-T. Nieh. Synthesis of nonzero clock skew circuits. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 25(6):961–976, June 2006.

[31] J.P. Fishburn. Clock skew optimization. *IEEE Trans. Comput.*, 39(7):945–951, July 1990.

[32] lp_solve 4.0. ftp://ftp.ics.ele.tue.nl/pub/lp_solve/.

[33] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. SIS: a system for sequential circuit synthesis. Memorandum No.UCB/ERL M92/41, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, May 1992.

[34] H.T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, January 1982.

[35] R. Roy and M.A. Bayoumi. An efficient two's complement systolic multiplier for real-time digital signal processing. *IEEE Trans. Circuit and Systems*, 36(11):1488–1493, November 1989.

[36] K. Obata, K. Takagi, and N. Takagi. A systolic SFQ divider using the radix-2 signed-digit representation. In *Proc. of the 2007 IEICE General Conference*, C-8-11, 2007. (in Japanese).

[37] M. Tanaka, K. Obata, K. Takagi, and N. Takagi. Design of a systolic square rooter using the signed-digital representation based on the single-flux-quantum logic. In *Proc. of the 2007 IEICE Society Conference*, C-8-9, 2007. (in Japanese).

Acknowledgment

I would like to express my sincere appreciation to Professor Naofumi Takagi of Nagoya University for his continuous guidance, interesting suggestions, accurate criticism and encouragements throughout this study. He introduced me to this study field.

I would also like to express my appreciation to Professor Hiroaki Takada of Nagoya University for his guidance in preparing the dissertation and for his helpful comments.

I would also like to express my appreciation to Professor Kazuyoshi Takagi of Nagoya University for his continuous guidance, interesting comments and encouragements. He assisted me to use various tools, software and computers.

I would also like to express my thanks to Dr. Masamitsu Tanaka of Nagoya University who gave me the basis of SFQ circuits. He also assisted me to design and test of SFQ integer multipliers in Chapter 6.

I would also like to express my thanks to Professor Akira Fujimaki of Nagoya University and all the member of his laboratory for their discussions and supports of this study.

I would also like to express my thanks to Professor Nobuyuki Yoshikawa of Yokohama National University and all the member of his laboratory for their discussions and supports of this study.

I would like to express my thanks to Dr. Yoshio Kameda, Dr. Shinichi Yorozu and Mr. Yoshihito Hashimoto of the Superconductivity Research Laboratory and Dr. Hirotaka Terai of the National Institute of Information and Communications Technology for their discussions and supports of this study.

I would like to thank Dr. Kazuhiro Nakamura for his support of my laboratory life at Nagoya University.

Thanks are also due to all the members of Professor Naofumi Takagi's Laboratory for their discussions and supports through this study.

List of Publications by the Author

Major Publications

1. K. Obata, K. Takagi, and N. Takagi, “Design method of dual-rail RSFQ logic circuits using 2×2-Join,” *IEICE Trans. Electron.*, vol.J88-C, no.3, pp.202–209, March 2005 (in Japanese).

2. K. Obata, M. Tanaka, Y. Tashiro, Y. Kamiya, N. Irie, K. Takagi, N. Takagi, A. Fujimaki, N. Yoshikawa, H. Terai, S. Yorozu, “Single-flux-quantum integer multiplier with systolic array structure,” *Physica C*, vol.445–448, pp.1014–1019, July 2006.

3. K. Obata, K. Takagi, and N. Takagi, “Logic synthesis method for dual-rail RSFQ digital circuits using root-shared binary decision diagrams,” *IEICE Trans. Fundamentals*, vol.E90-A, no.1, pp.257–266, Jan. 2007.

4. K. Obata, K. Takagi, and N. Takagi, “A method of sequential circuit synthesis using one-hot encoding for single-flux-quantum digital circuits,” *IEICE Trans. Electron.*, vol.E90-C, no.12, pp.2278–2284, Dec. 2007.

5. K. Obata, K. Takagi, and N. Takagi, “A clock scheduling algorithm for high-throughput RSFQ digital circuits,” submitted to *IEICE Trans. Fundamentals.*

Technical Reports

1. K. Obata, K. Takagi, and N. Takagi, “Logic design method of dual-rail RSFQ digital circuits using decision diagrams,” In *LA Symposium 2004 Winter*, pp.20.1–20.6, Feb. 2004 (in Japanese).

Convention Records

1. K. Obata, K. Takagi, and N. Takagi, “Design of RSFQ adders and comparators using dual-rail logic,” In *Proc. of the 2003 IEICE Society Conference*, C-8-11, 2003 (in Japanese).

2. K. Obata, K. Takagi, and N. Takagi, “Logic design method for dual-rail RSFQ circuits using binary decision diagrams,” In *Proc. of the 2004 IEICE General Conference*, C-8-1, 2004 (in Japanese).

3. K. Obata, Y. Tashiro, M. Tanaka, T. Kawamoto, Y. Kamiya, K. Takagi, N. Takagi, and A. Fujimaki, “Serial-parallel two’s complement multiplier for SFQ circuit implementation,” In *Proc. of the 2005 IEICE General Conference*, C-8-4, 2005 (in Japanese).

4. K. Obata, K. Takagi, and N. Takagi, “Clock tree synthesis for synchronous clocking SFQ circuits,” In *Proc. of the 2005 IEICE Society Conference*, C-8-9, 2005 (in Japanese).

5. K. Obata, M. Tanaka, Y. Tashiro, Y. Kamiya, N. Irie, K. Takagi, N. Takagi, A. Fujimaki, H. Terai, and S. Yorozu, “Single-flux-quantum integer multiplier with systolic array structure,” In *18th International Symposium on Superconductivity (ISS)*, October 24-26, 2005, Tshukuba, Japan.

6. K. Obata, K. Takagi, and N. Takagi, “A design method of sequential circuits with

one-hot encoding for single-flux-quantum digital circuits," In *Proc. of the 2006 IEICE Society Conference*, A-3-10, 2006 (in Japanese).

7. K. Obata, K. Takagi, and N. Takagi, "A systolic SFQ divider using the radix-2 signed-digit representation," In *Proc. of the 2007 IEICE General Conference*, C-8-11, 2007 (in Japanese).

8. K. Obata, T. Furuta, K. Takagi, and N. Takagi, "A high-throughput SFQ bit-serial floating-point multiplier based on systolic architecture," In *Proc. of the 2007 IEICE Society Conference*, C-8-10, 2007 (in Japanese).