國立臺灣大學電機資訊學院電子工程學研究所
碩士論文
Graduate Institute of Electronics Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

邏輯電路摺疊：由組合電路轉換至序向電路

Circuit Folding: From Combinational to Sequential Circuits

錢柏均
Po-Chun Chien

指導教授：江介宏 博士
Advisor: Jie-Hong Roland Jiang, Ph.D.

中華民國 109 年 6 月
June, 2020

doi:10.6342/NTU202001058

國立臺灣大學碩士學位論文
口試委員會審定書

邏輯電路摺疊：由組合電路轉換至序向電路
Circuit Folding: From Combinational to Sequential Circuits

本論文係錢柏均君（R07943091）在國立臺灣大學電子工程學研究所完成之碩士學位論文，於民國 109 年 5 月 6 日承下列考試委員審查通過及口試及格，特此證明

口試委員：江介宏

李宏毅（指導教授）　王振宇

陳郁方　顏嘉志

系主任、所長 林宗男

Acknowledgements

首先，我想感謝我的指導教授江介宏老師。自大學時期專題研究以來，以及研究所的這兩年的時間裡，教會了我嚴謹的科學方法，以及如何清楚的闡述自己的想法並寫成論文；在我研究遇到瓶頸時也會與我一同討論，尋找解決方法，並給予許多方向上的建議。這幾年合作經驗告訴我江教授是一位認眞、謙虛且內斂的學者，有了他的指導，我才能順利的成這篇論文。

接著，我想感謝實驗室的學長們的幫助，念澤、鬍子及鶴騰學長提供了研究上的建議，子鈞、韋智及家志學長分享了研究生及課業上的經驗，奕凡學長推薦了台大附近聚餐首選餐廳。另外也得感謝實驗室和我一起打拼的同伴們，彥廷及裕洲，不但課業上互相扶持，在研究上提供了需許多援助，也幫助我在PS4控制上得到了微幅的技術提升。總之，謝謝ALCom的所有人，在兩年的研究所期間，一同欣賞了不少影視巨作（如在幻海奇情中所展現的頑強求生意志至今仍歷歷在目，著實令人動容），讓平時稍顯枯燥的生活中，增添了不少色彩。

另外，我也想感謝蔡益坤老師、王柏堯老師、陳郁方老師、顏嘉志博士願意擔任我的口試委員，並給予了我許多寶貴的意見。這篇論文能獲得眾位學者的肯定，是我的榮幸。

最後，我要感謝我的家人及女友，給予了我許多的支持與肯定，讓我能無後顧之憂地完成學業。

錢柏均

Po-Chun Chien

National Taiwan University

May 8th, 2020

doi:10.6342/NTU202001058

邏輯電路摺疊：由組合電路轉換至序向電路

研究生: 錢柏均　　指導教授: 江介宏　　博士

國立臺灣大學電子工程學研究所

摘要

在這篇論文中，我們制訂了時框展開（time-frame expansion）之逆操作——時框摺疊（time-frame folding）。 時框展開爲一常用於自動測試圖樣產生及模型檢查之技術，它會將一序向邏輯電路展開成一組和邏輯電路；而時框摺疊則會進行反方向的操作，但由於每個時框下的分支電路都可能是不同的，因此它是一個相當複雜的技術。 時框摺疊可應用於測試平台生成及有界策略一般化的領域中。 我們提出的演算法可以找到一個最小的有限狀態機，有著與欲折疊的電路相同的輸入/輸出行爲表現。 再者，我們將時框摺疊延伸爲功能性電路摺疊，並另外提出了結構性電路折疊。 藉由上述兩個電路摺疊技術，我們可以於現場可程式化邏輯閘陣列（FPGA）中達到分時多工之效果，以解決FPGA中輸入及輸出接腳不足的瓶頸。 大多現有的研究是以實體設計的角度去解決此瓶頸，並設法藉由電路分割或繞線去減少切點網路之數量。 我們所提出的方法以不同的角度切入，並可以在帶寬及通量兩者間提供自由的取捨。 實驗的結果顯示出時框摺疊有著電路簡化的能力；同時也展現了結構性電路摺疊之效力及可拓展性，以及功能性電路摺疊之優化能力，幫助我們得到更少的查找表及正反器之電路形式。

關鍵字: 邏輯電路摺疊、函式分解、接腳數簡化、有限狀態自動機最小化、時框展開、時框摺疊、分時多工

doi:10.6342/NTU202001058

Circuit Folding: From Combinational to Sequential Circuits[1]

Student: Po-Chun Chien Advisor: Jie-Hong Roland Jiang Ph.D.

Graduate Institute of Electronics Engineering

National Taiwan University

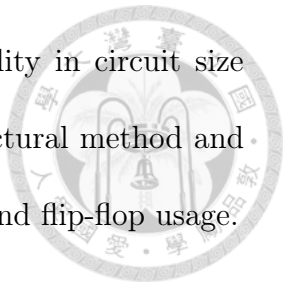
Abstract

In the thesis, we formulate time-frame folding (TFF) as the reverse operation of time-frame expansion in automatic test pattern generation (ATPG) and (un)bounded model checking. While the latter converts a sequential circuit into a combinational one for some expansion bound of k time-frames, the former attempts the opposite, which can be highly non-trivial as the subcircuit of each time-frame can be distinct. TFF arises naturally in the context of testbench generation and bounded strategy generalization. We propose an algorithm that finds a minimum-state finite state machine consistent with the input-output behavior of the combinational circuit under folding. Furthermore, we extend TFF as functional circuit folding and introduce structural circuit folding. Through the two folding methods, we formulate a new approach at the logic level to achieve time multiplexing, which is an important technique to overcome the bandwidth bottleneck of limited input-output pins in FPGAs. Most prior work tackles the problem of time multiplexing from a physical design standpoint to minimize the number of cut nets or Time Division Multiplexing (TDM) ratio through circuit partitioning or routing. Our formulation is orthogonal to the previous ones and provides a smooth trade-off between bandwidth and

[1]This thesis is the extension research published in [11, 12]

doi:10.6342/NTU202001058

throughput. Empirical evaluation of TFF demonstrates its ability in circuit size compaction. Experiments also show the effectiveness of the structural method and improved optimality of the functional method on look-up-table and flip-flop usage.



Keywords: circuit folding, functional decomposition, pin-count reduction, state minimization, time-frame expansion, time-frame folding, time multiplexing

doi:10.6342/NTU202001058

Contents

Verification Letter from the Oral Examination Committee i

Acknowledgements ii

Chinese Abstract iii

Abstract iv

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Time-frame Folding 2

1.2 Time Multiplexing 3

1.3 Our Contributions 6

1.4 Thesis Organization 7

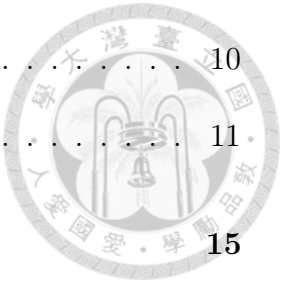
2 Preliminaries 8

2.1 Finite State Machine 9

2.2 Combinational Circuit 9

2.3 Sequential Circuit 9

doi:10.6342/NTU202001058



2.4 Time-frame Expansion 10
2.5 Functional Decomposition 11

3 Time-frame Folding **15**

3.1 Problem Formulation 15
3.2 Algorithm 16
3.2.1 State Identification via Functional Decomposition 17
3.2.2 Transition Reconstruction 19
3.2.3 State Minimization 21
3.2.4 State Encoding 23
3.3 Implementation Issues 23

4 Circuit Folding for Time Multiplexing **25**

4.1 Problem Formulation 25
4.2 Structural Circuit Folding 26
4.3 Functional Circuit Folding 29
4.3.1 Pin Scheduling and Iterative Circuit Conversion 29
4.3.2 FSM Construction via Time-Frame Folding 32
4.3.3 FSM Minimization 33
4.3.4 FSM Encoding 33

5 Experiments **35**

5.1 Time-frame Folding 35
5.1.1 Fixed Point after TFF 38
5.1.2 Circuit Size Compaction 41
5.2 Time Multiplexing via Circuit Folding 42

doi:10.6342/NTU202001058

5.2.1 Structural Folding on Large Circuits 44
5.2.2 Comparing Structural and Functional Folding 48
5.2.3 Case Study of Combining Structural and Functional Folding . 52

6 Conclusions and Future Work **56**

6.1 Conclusions 56
6.2 Future Work 57

Bibliography **59**

doi:10.6342/NTU202001058

List of Figures

1.1 Time-frame expansion vs. folding. 2

1.2 An illustration of multi-FPGA prototyping system. 4

1.3 The ratio FPGA logic capacity over I/Os (retrieved from [32]). 4

1.4 TDM I/O transmission with ratio 4. 5

2.1 Sequential circuit `s27`. 11

2.2 Time-frame expanded circuit of `s27`. 12

2.3 Effect of functional decomposition. 13

2.4 BDD-based functional decomposition. 14

3.1 Computation flow of time-frame folding. 16

3.2 State identification. 19

3.3 State transition graphs of FSMs. 22

4.1 Illustration of structural circuit folding. 27

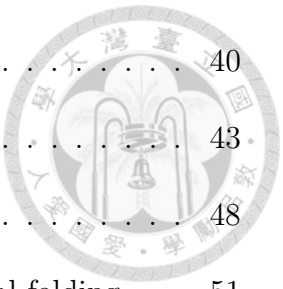
4.2 Example of 3-bit adder (`3-adder`) circuit under folding. 28

4.3 Computation flow of functional circuit folding. 30

4.4 FSM by functional circuit folding of `3-adder`. 34

5.1 #state vs. #time-frame. 39

doi:10.6342/NTU202001058



5.2 Total runtime vs. #time-frame. 40

5.3 Circuit size after TFF. 43

5.4 Circuit size after structural folding. 48

5.5 Circuit size comparison between structural and functional folding. . . 51

5.6 Hierarchical structure of `C7552`. 53

doi:10.6342/NTU202001058

List of Tables

5.1 Results of TFF on ISCAS and ITC benchmarks. 37

5.2 Results of time-frame folding on homing sequence benchmarks. 38

5.3 Results on folding with fixed point. 41

5.4 Benchmark statistics. 43

5.5 Results of structural circuit folding. 47

5.6 Comparison between structural and functional methods. 50

5.7 Results of folding adders. 55

5.8 Results of folding voters. 55

5.9 Results of folding `C7552` with the structural and functional methods combined. 55

doi:10.6342/NTU202001058

Chapter 1

Introduction

Circuit folding is a process of transforming a combinational circuit $\mathcal{C}_C$ into a sequential circuit $\mathcal{C}_S$, which after time-frame expansion, becomes functionally equivalent to $\mathcal{C}_C$. The computation of the combinational circuit is folded into multiple iterations of the resulting sequential circuit. With such folding process, one can reduce the I/O pint count of the circuit and potentially lower the overall complexity. Circuit folding finds its applications testbench generation and time multiplexing in multi-FPGA systems, which are crucial to the field of logic synthesis, and yet remains relatively unstudied. In this chapter, we introduce the motivation and the application field of the thesis in Section 1.1 and 1.2, highlight our contributions in Section 1.3, and provide the overall thesis structure in Section 1.4.

doi:10.6342/NTU202001058

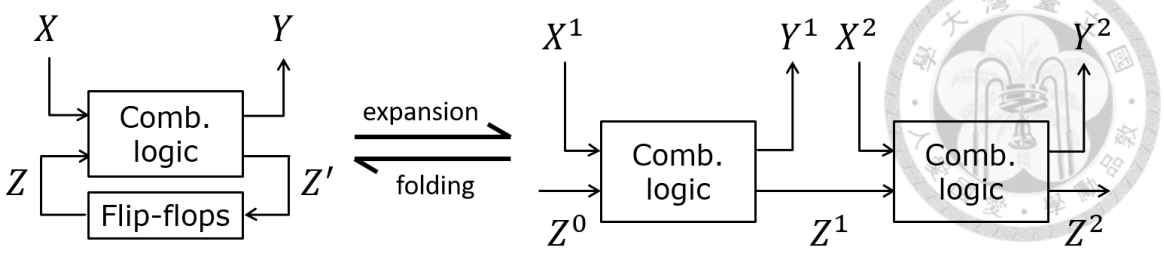
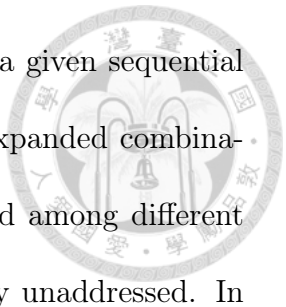


Figure 1.1: Time-frame expansion vs. folding.

1.1 Time-frame Folding

Time-frame folding (TFF) is the reverse operation of time-frame unfolding (TFU), or time-frame expansion as illustrated in Figure 1.1. While TFU is a well-known technique commonly used in, e.g., automatic test pattern generation (ATPG) [36] and (un)bounded model checking of sequential circuits [5], TFF remains largely unstudied. In fact, TFF finds its natural applications. For example, to test a sequential design, one may look for a testbench that produces some set of desired test patterns of length-bounded input-output sequences. The testbench can be represented directly by a large combinational circuit, corresponding to a time-frame expanded version of a sequential circuit, or represented more compactly by a sequential circuit. For another example, in model-based testing of software systems [19, 30], in state identification [21], and in system initialization [28], one may be asked to compute (non-adaptive or adaptive) homing, distinguishing, and/or synchronizing sequences. These problems can be formulated as quantified Boolean formula (QBF) [6] solving of strategy derivation, e.g., in [35], that computes the intended sequence. Again, the homing, distinguishing, or other strategies under synthesis can be represented directly by a large combinational circuit or more compactly by a sequential circuit.

doi:10.6342/NTU202001058



However, unlike the straightforward derivation of TFU from a given sequential circuit, TFF can be highly non-trivial because the time-frame expanded combinational circuit may not exhibit a common circuit structure shared among different time-frames. Perhaps it is this difficulty that makes TFF largely unaddressed. In this work, we formulate the TFF problem and provides a general solution that makes no structure assumption on the combination circuit under time-frame folding.

To the best of our knowledge, this work is the first to address the time-frame folding issue. Most related prior work on time-frame issues centered around unfolding, e.g. in [24]. While the prior work converts a sequential circuit into a combinational one with respect to some expansion bound k time-frames, our attempt is the opposite. Regarding our method, we rely on multiple-output functional decomposition [18] to identify equivalent states as part of our computation flow. A similar technique has been applied in sequential equivalence checking [17].

1.2 Time Multiplexing

The concept of folding is then further extended for general combinational circuits to tackle the time multiplexing problem in FPGAs. Multi-FPGA boards are commonly used for system emulation [25] and prototyping as illustrated in Figure 1.2. As the logic capacity, i.e., the number of look-up-tables (LUTs), of an FPGA increases with new technology nodes, the growth in I/O pin count remains relatively slow. The ratio of FPGA logic capacity over I/O over the past few years is plotted in Figure 1.3. This unbalance growth rate makes the number of available I/O pins for each FPGA relatively small compared to the number of required inter-chip signals,

doi:10.6342/NTU202001058

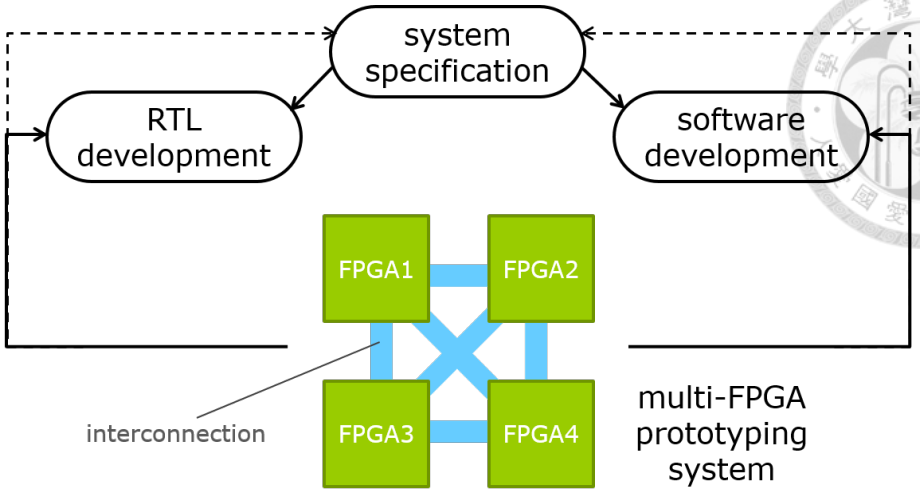


Figure 1.2: An illustration of multi-FPGA prototyping system.

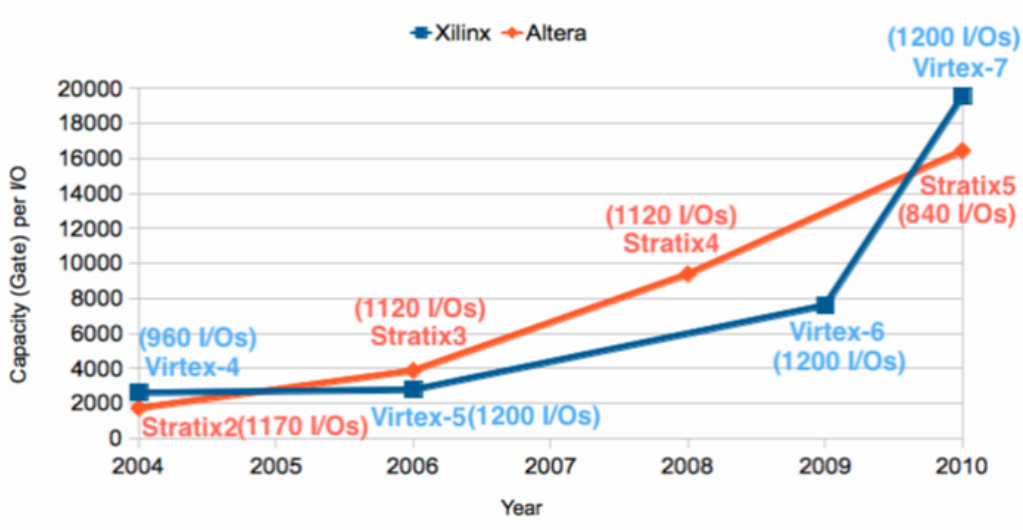


Figure 1.3: The ratio FPGA logic capacity over I/Os (retrieved from [32]).

which leads to a significant underutilization of logic resources [16, 32].

To overcome the bottleneck of limited inter-chip I/O bandwidth, time division multiplexing (TDM) [4] was proposed, where physical pins and wires are multiplexed among multiple signals, increasing the effective number of available logic pins. Under this scheme, the system requires two separate clocks, a system clock, on which the FPGAs operate, and a faster I/O clock, on which the inter-chip signals are propagated. The ratio of the system clock to the I/O clock is called the TDM ratio r.

doi:10.6342/NTU202001058

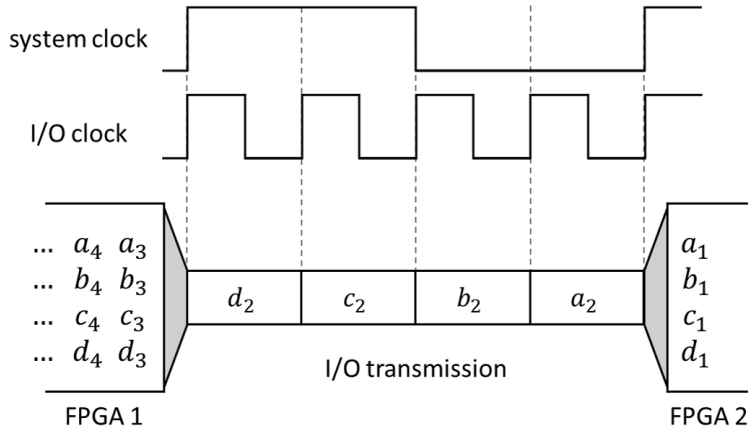
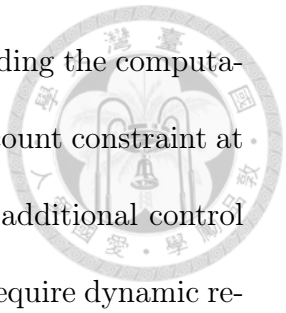


Figure 1.4: TDM I/O transmission with ratio 4.

Essentially, r times the I/O bandwidth of signals can be transmitted during a system clock. Figure 1.4 illustrates an example of I/O transmission between two FPGAs with TDM ratio 4. The TDM technique dramatically increases the capability of multi-FPGA systems. However, it reduces the system throughput as the system clock is operating at a lower frequency. Most of the related work, e.g., [9], viewed this problem from a physical design standpoint and tried to minimize the number of cut nets, which corresponds to the number of inter-chip signals, passing through each FPGA. Another line of research, e.g., [22,33], considers scheduling and temporal partitioning for time-multiplexed FPGAs. They partitioned a combinational circuit into several pipeline stages for time multiplexing. However, the approach cannot control the pin-count reduction as it is determined by the circuit structure. In [15,23], the problem of pin assignment during pin multiplexing, which is the mapping between logic inputs and outputs to the physical pins, was investigated. The pin-count reduction issue was not addressed.

In this work, we formulate a new orthogonal approach to achieve time multiplexing at the logic level. The proposed structural and functional methods can directly

doi:10.6342/NTU202001058



reduce the number of input pins of a logic circuit as desired by folding the computation of the circuit. The resulting circuit will satisfy the input pin count constraint at the cost of additional flip-flops storing required information and additional control circuitry for intended computation. This new approach does not require dynamic reconfiguration of the FPGA, unlike [22,33]. Neither does it require an additional I/O clock as TDM, the I/O transmission can work in synchronization with the system clock.

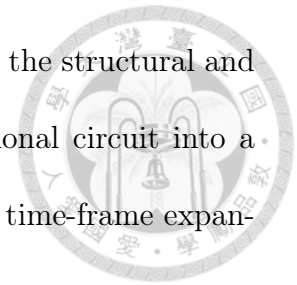
In the literature, the term "folding" is used elsewhere. In [27], a folding transformation technique was proposed to schedule and bind a data-flow graph onto a hardware architecture, where folding refers to the process of executing multiple algorithmic operations in a hardware unit. In [14], a folding technique was proposed to identify structurally identical subcircuits to share gate implementation using dual-edge-triggered flip-flops for time multiplexing. Their primary objective was to minimize the circuit area after technology mapping, while ours is to reduce the input pin count.

1.3 Our Contributions

This thesis is the extension research published in [11, 12]. The main results of this work include:

1. We motivate and formulate the problem of time-frame folding, and propose an algorithm that finds a minimum-state finite state machine consistent with the input-output behavior of the combinational circuit under folding.

doi:10.6342/NTU202001058



2. We formulating a new time multiplexing scheme, and propose the structural and functional circuit folding methods, that convert a combinational circuit into a sequential one with equivalent input-output behavior modulo time-frame expansion.

3. We evaluate the proposed TFF algorithm and show the computational viability and its ability in circuit size compaction for potential use in different application domains. We conduct experiments on the proposed structural and functional folding methods, and demonstrate their effectiveness in input pin reduction to alleviate the I/O pin bottleneck of FPGAs.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides the essential preliminaries. In Chapter 3, we formulate the problem of time-frame folding and present our algorithmic solution. The problem of time multiplexing is then formulated in Chapter 4, along with the details of the proposed structural and functional methods. Chapter 5 evaluates the experimental results, and finally Section 6 concludes this thesis.

doi:10.6342/NTU202001058

Chapter 2

Preliminaries

In the sequel, sets are denoted by upper-case letters, e.g. S; the elements in a set are in lower-case letters, e.g. $a \in S$; the cardinality of a set S is denoted as $|S|$. A partition P of a set S into non-empty subsets $S_i \subseteq S$, for $i = 1, \ldots, k$, is denoted by $P = \{S_1|S_2|...|S_k\}$, where $S_i \cap S_j = \emptyset, \forall i \neq j$ and $\bigcup_i S_i = S$. Each S_i is a called a *cell* of P. Let P and P' be two partitions of a set S. Partition P is said to be a *refinement* of P', if $s_i, s_j \in S$ are in different cells of P', then $s_i, s_j \in S$ are in different cells of P. Note that the refinement relation is not symmetric, i.e., that P is a refinement of P' does not imply that P' is a refinement of P. For a set of Boolean variables X, its set of truth assignments is denoted by $[\![X]\!]$, e.g., $[\![X]\!] = \{(0,0),(0,1),(1,0),(1,1)\}$ for $X = \{x_1, x_2\}$. Boolean negation, conjuction, and disjunction are denoted by $\neg$ or overline, $\wedge$ or $\cdot$, and $\vee$ or $+$, respectively.

doi:10.6342/NTU202001058

2.1 Finite State Machine

A finite state machine (FSM) can be described by a six-tuple $(I, O, Q, q_1, \Delta, \Omega)$, where I is the input alphabet, O is the output alphabet, $Q \neq \emptyset$ is a non-empty finite set of states, $q_1 \in Q$ is the initial state, $\Delta : Q \times I \to Q$ is the state transition function, $\Omega : Q \times I \to O$ is the output function. A machine is completely specified if, for every state in Q under every input, its output and next state are defined; otherwise, it is incompletely specified. An FSM can be alternatively represented as a state transition graph.

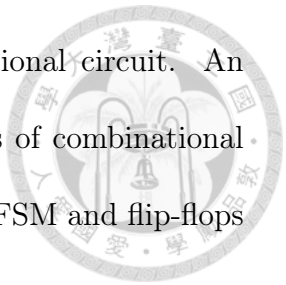
2.2 Combinational Circuit

A combinational circuit $\mathcal{C}_C$ is a directed acyclic graph with vertices V and edges $E \subseteq V \times V$. Two subsets $I, O \subset V$ are identified as the *primary inputs* (PIs) and *outputs* (POs), respectively. For $(u, v) \in E$, we call u is a *fanin* of v, and v is a *fanout* of u. Each vertex $v \in V$ is associated with a Boolean variable and with a Boolean function expressed in terms of its fanin variables. The *support set* of v is the set of PIs that can reach v through a path consisting of edges in E.

2.3 Sequential Circuit

A sequential circuit $\mathcal{C}_S$ is a combinational circuit augmented with state-holding elements (flip-flops), each of which takes an output of the combinational circuit as

doi:10.6342/NTU202001058



its input and produces an output to an input of the combinational circuit. An FSM can be implemented by a sequential circuit, which consists of combinational logic netlists realizing the transition and output functions of the FSM and flip-flops holding current state values.

2.4 Time-frame Expansion

The operation of a sequential circuit can be seen as an iterative combinational circuit that repeats the same computation but taking timestamped inputs. In time-frame expansion/unfolding, a sequential circuit is unrolled to construct an iterative combinational circuit. This is done by cascading duplicated sequential circuits, where the input and output of the flip-flops in the adjacent time-frames are connected together. In this paper, the initial values of the flip-flops (initial state) is constant-propagated throughout the time-frames. Therefore, after expansion, the primary output functions of each time-frame in the expanded circuit can be viewed as a purely combinational logic which depends on the primary inputs of all the previous time-frames.

Example 2.1 Figure 2.1 shows the circuit structure of `s27`, where x_i denotes the i^{th} primary input variable, y denotes the primary output variable, and z_i and z_i' denote the current- and next-state variables, respectively, of the i^{th} flip-flop. Let v^t denote the variable v instantiated at the t^{th} time-frame. Figure 2.2a shows the circuit of `s27` after three time-frames of expansion, and Figure 2.2b shows the same circuit after simplification with constant propagation of the initial state values $(z_1^0, z_2^0, z_3^0) = (0, 0, 0)$. Note that after the time-frame expansion all primary output

doi:10.6342/NTU202001058

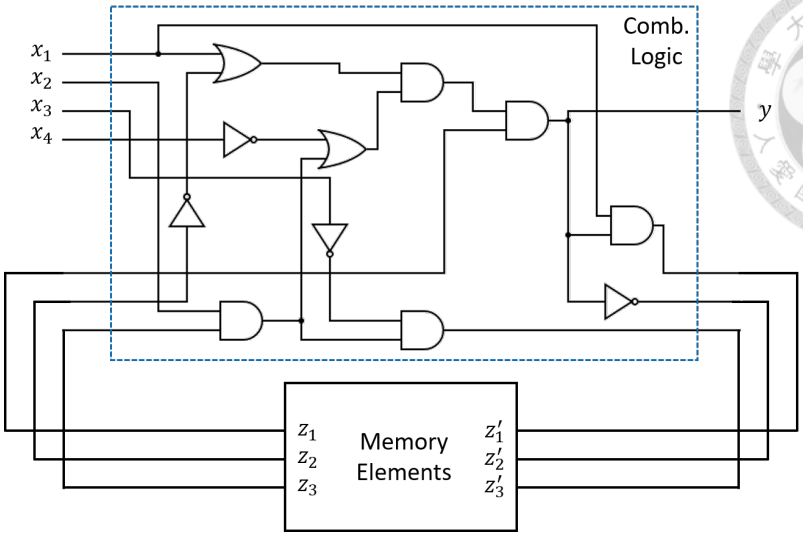


Figure 2.1: Sequential circuit `s27`.

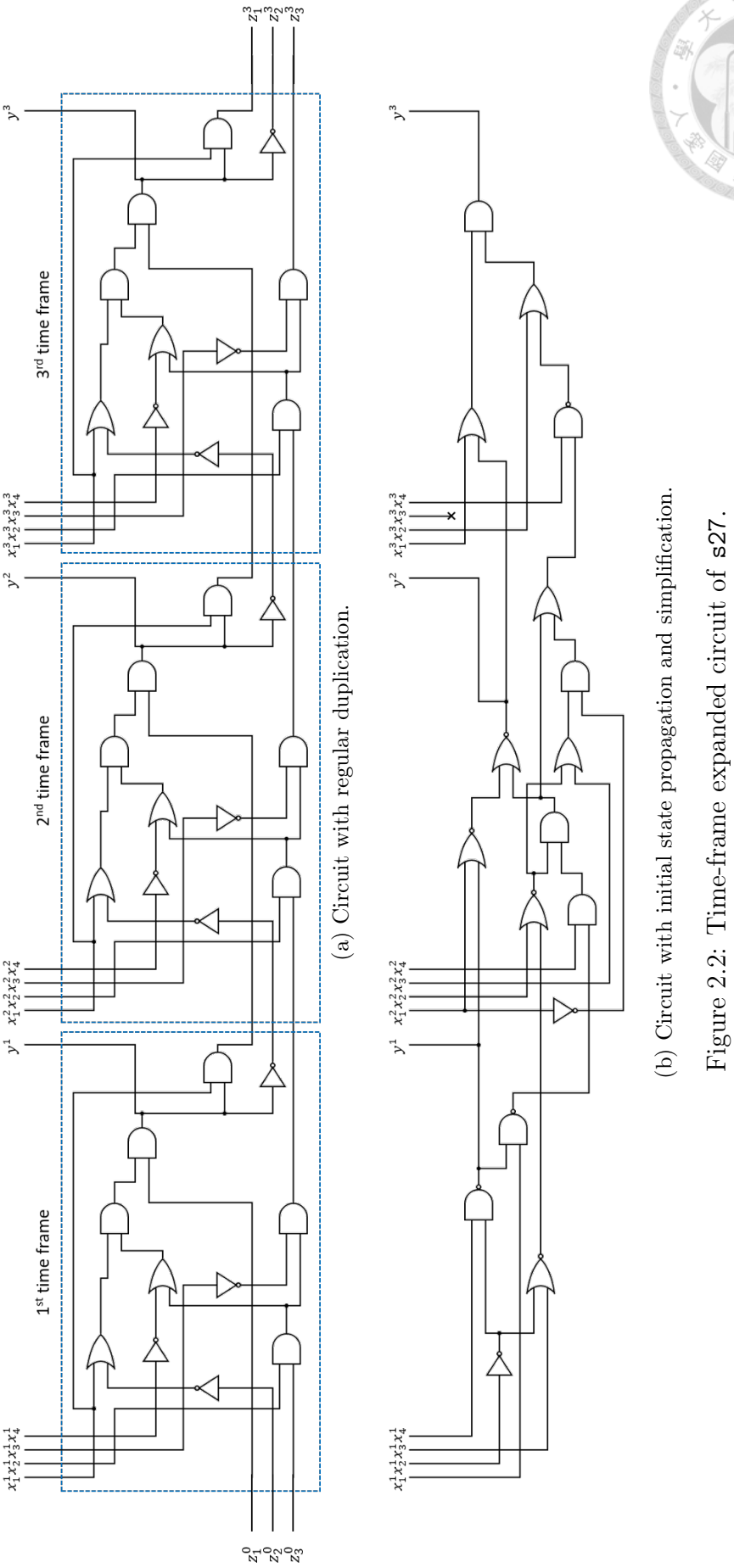
functions are purely combinational, and after further circuit simplification the state transition functions cannot be clearly identified. In the following, the timestamp of the input/output of an iterative circuit is denoted by superscript letters. E.g., for a sequential circuit with input x and output y, the timestamped input and output of the iterative circuit is denoted as $x^1, \ldots, x^t$ and $y^1, \ldots, y^t$, respectively.

2.5 Functional Decomposition

Given a single-output Boolean function $f(X)$, the *functional decomposition* [3, 29] problem asks to re-express $f(X) = f_\mu(X_\mu, f_{\lambda_1}(X_\lambda), \ldots, f_{\lambda_k}(X_\lambda))$, where X_λ and X_μ are called the *bound set* and *free set* variables, respectively, which form a partition on $X = \{X_\lambda | X_\mu\}$.[2] Let $F_\lambda(X_\lambda) = \{f_{\lambda_1}(X_\lambda), \ldots, f_{\lambda_k}(X_\lambda)\}$. To avoid trivial decomposition, it is required that $|F_\lambda| < |X_\lambda|$. Figure 2.3 illustrates the structural effect of functional decomposition.

[2]In time-frame folding application, only disjoint decomposition, i.e., $X_\lambda \cap X_\mu = \emptyset$, needs to be considered.

doi:10.6342/NTU202001058



(a) Circuit with regular duplication.

(b) Circuit with initial state propagation and simplification.

Figure 2.2: Time-frame expanded circuit of s27.

doi:10.6342/NTU202001058

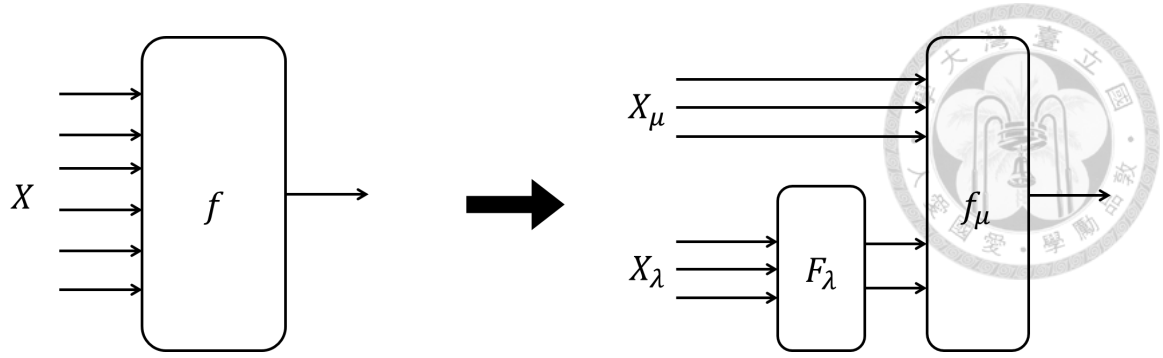


Figure 2.3: Effect of functional decomposition.

Functional decomposition can be defined for multiple single-output functions $f_1(X), \ldots, f_m(X)$, and considered as decomposing a multiple-output function $F(X) = (f_1(X), \ldots, f_m(X))$. In [18], a technique called *hyperfunction* encoding is introduced to encode a multiple-output function into a single-output function with $\lceil \log_2 |F| \rceil$ auxiliary pseudo input variables. E.g., for $m = 4$, two auxiliary variables $A = \{\alpha_1, \alpha_2\}$ can be used to build the hyperfunction $h(X, A) = \neg\alpha_1\neg\alpha_2 f_1(X) + \neg\alpha_1\alpha_2 f_2(X) + \alpha_1\neg\alpha_2 f_3(X) + \alpha_1\alpha_2 f_4(X)$. Thereby, a single-output functional decomposition algorithm can be applied to decompose a multiple-output function.

Functional decomposition can be achieved based on the reduced ordered binary decision diagram (ROBDD) [8, 20]. In BDD-based decomposition, the ROBDD of the function $f(X)$ under decomposition is built with the variable ordering constraint that the bound set variables X_λ are ordered above the free set variables X_μ. The *cut set* of the ROBDD is the set of BDD nodes controlled by free set variables that are pointed to by some edge from a node controlled by a bound set variable. Essentially, for c being the cut set size, then $|F_\lambda| \geq \lceil \log_2 c \rceil$.

Example 2.2 Figure 2.4 shows the BDD-based decomposition for function y^2 of the time-frame expanded circuit `s27`. The cut set $\{n_1, n_2, n_3\}$ is induced by setting $X_\lambda = \{x_1^1, x_2^1, x_3^1, x_4^1\}$ and $X_\mu = \{x_1^2, x_2^2, x_4^2\}$. Necessarily two bits are needed to

doi:10.6342/NTU202001058

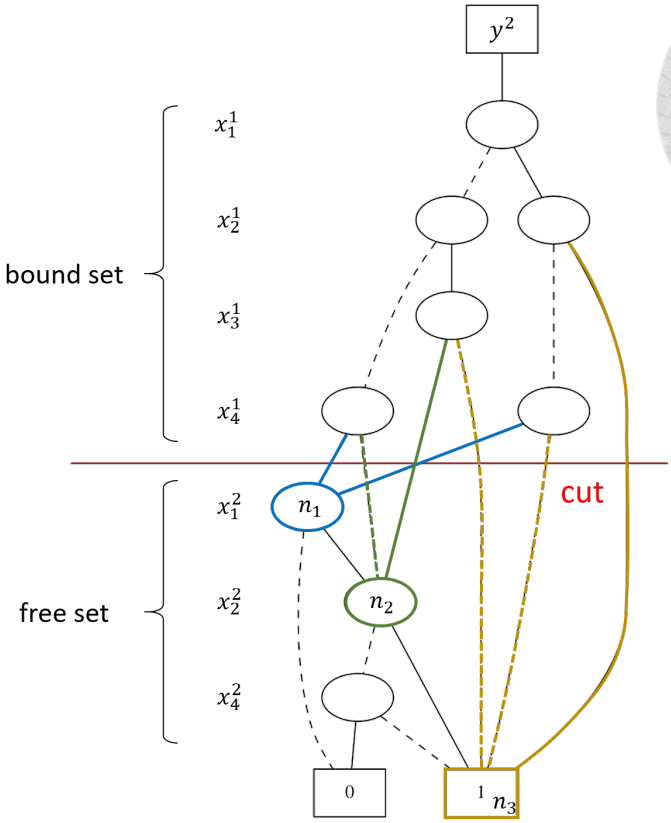


Figure 2.4: BDD-based functional decomposition.

re-encode the bound set variables to distinguish the three cut set nodes. Hence, $|F_\lambda| \geq 2$.

doi:10.6342/NTU202001058

Chapter 3

Time-frame Folding

Time-frame folding is the reverse operation of time-frame unfolding or time-frame expansion. In this chapter, we describe the derivation of the minimum-state FSM from folding an iterative (time-frame expanded) combinational circuit. The chapter is organized as follows. The problem of time-frame folding is formulated in Section 3.1. Our algorithmic solution is then presented in Section 3.2, and implementation improvement in Section 3.3.

3.1 Problem Formulation

The problem of *time-frame folding* can be stated as follows.

Problem Statement 3.1 (Time-Frame Folding)

Given a k-iterative combinational circuit $\mathcal{C}_C$ with inputs $X^1, \ldots, X^k$ for $X^t = \{x_1^t, \ldots, x_n^t\}$ and outputs $Y^1, \ldots, Y^k$ for $Y^t = \{y_1^t, \ldots, y_m^t\}$, find a sequential cir-

doi:10.6342/NTU202001058

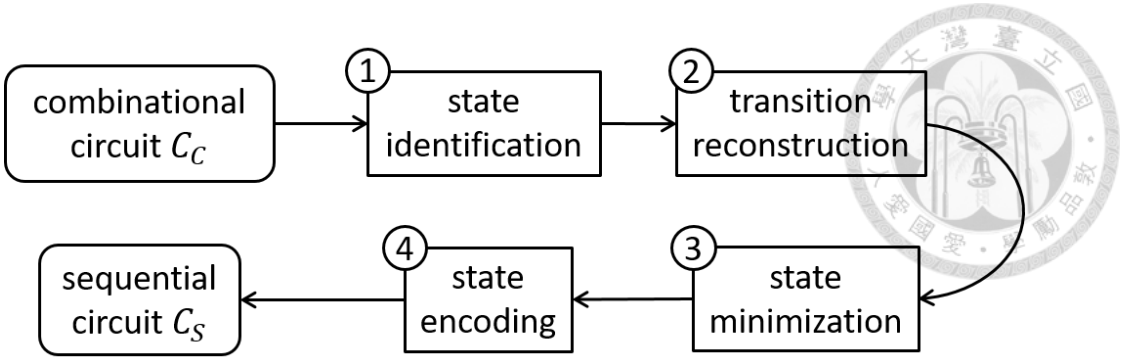


Figure 3.1: Computation flow of time-frame folding.

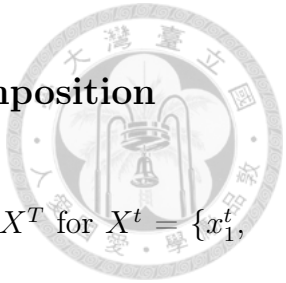
cuit $\mathcal{C}_S$ with inputs $X = \{x_1, \ldots, x_n\}$ and outputs $Y = \{y_1, \ldots, y_m\}$ such that the input-output behavior of $\mathcal{C}_S$ within the first k time-frames is the same as that of $\mathcal{C}_C$. Moreover, the number of states of $\mathcal{C}_S$ is minimized.

Note that the statement makes no assumption on the circuit structure of $\mathcal{C}_C$ but only its inputs and outputs in an iterative form, crucial for time-frame folding.

3.2 Algorithm

The computation flow of the TFF algorithm is shown in Figure 3.1. Given as input an iterative combinational circuit $\mathcal{C}_C$ with inputs X^1, ..., X^T for $X^t = \{x_1^t, \ldots, x_n^t\}$ and outputs Y^1, ..., Y^T for $Y^t = \{y_1^t, \ldots, y_m^t\}$, the algorithm returns a sequential circuit with inputs $X = \{x_1, \ldots, x_n\}$ and outputs $Y = \{y_1, \ldots, y_m\}$ consistent with $\mathcal{C}_C$ in T time-frames. It consists of the following steps: 1) state identification by functional decomposition, 2) state transition reconstruction, 3) state minimization, and 4) state encoding. The steps are detailed in the following subsections.

doi:10.6342/NTU202001058



3.2.1 State Identification via Functional Decomposition

Given an iterative combinational circuit $\mathcal{C}_C$ with inputs $X^1, \ldots, X^T$ for $X^t = \{x_1^t, \ldots, x_n^t\}$ and outputs $Y^1, \ldots, Y^T$ for $Y^t = \{y_1^t, \ldots, y_m^t\}$, we show that the notion of states at time-frame t is induced by the output functions of $Y^{t+1}, \ldots, Y^T$. Note that the outputs Y^t observed at time t induce an equivalence relation on the set of input assignments $[\![X^1 \cup \ldots \cup X^t]\!]$. Effectively, the equivalence relation forms a partition on $[\![X^1 \cup \ldots \cup X^t]\!]$. Assume that the partition on $[\![X^1 \cup \ldots \cup X^t]\!]$ induced by the equivalence relation imposed by the outputs $Y^{t+1}, \ldots, Y^T$ has k cells (equivalence classes). Then we know the signals communicating from iteration t to iteration $t+1$ in circuit $\mathcal{C}_C$ (i.e., the information of inputs $X^1, \ldots, X^t$ needed to compute outputs $Y^{t+1}, \ldots, Y^T$) must have at least $\lceil \log_2 k \rceil$ bits. In the functional decomposition viewpoint of Figure 2.3, by decomposing the hyperfunction f of the output functions of $Y^{t+1} \cup \ldots \cup Y^T$ with bound set variables $X_\lambda = X^1 \cup \ldots \cup X^t$ and free set variables $X_\mu = X^{t+1} \cup \ldots \cup X^T \cup A$, where A is the set of pseudo input variables introduced to encode functions $Y^{t+1} \cup \ldots \cup Y^T$, the number of bits needed to communicate from F_λ to f_μ in the picture of Figure 2.3 is at least $\lceil \log_2 k \rceil$. Essentially the k equivalence classes correspond to the minimum states needed to distinguish the input assignments $[\![X^1 \cup \ldots \cup X^t]\!]$ for the outputs $Y^{t+1}, \ldots, Y^T$ to produce correct valuation. Let $Q^t = \{q_1^t, ..., q_k^t\}$ be the *states* representing the k equivalence classes, and let $\tau^t = \{\tau_{q_1^t}, \ldots, \tau_{q_k^t}\}$ be the set of *transition conditions*, that is, characteristic functions, each characterizing a set of equivalent input assignments in an equivalence class of $[\![X^1 \cup \ldots \cup X^t]\!]$. Then Q^t and τ^t can be obtained from ROBDD-based functional decomposition by noting that Q^t corresponds to the

doi:10.6342/NTU202001058

cut set and τ^t corresponds to the path conditions from the root node leading to the cut set nodes. In the sequel, we let $S^t = \{(q_1^t, \tau_{q_1^t}), \ldots, (q_k^t, \tau_{q_k^t})\}$ be the set of state and transition condition pairs at time t.

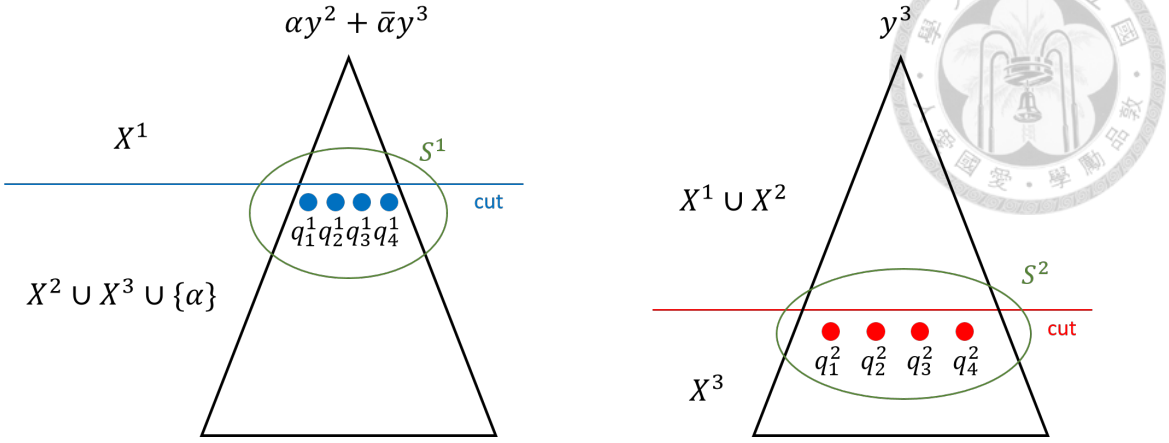
Example 3.1 To demonstrate how Q^t and τ^t are obtained from ROBDD-based functional decomposition, we take y^2 in Figure 2.4 as an example. To compute S^1, we build the hyperfunction $h = \alpha y^2 + \neg\alpha y^3$ of the output functions y^2 and y^3 as illustrated in Figure 3.2a. By performing functional decomposition on h, we obtain $S^1 = \{(q_1^1, \tau_{q_1^1}), (q_2^1, \tau_{q_2^1}), (q_3^1, \tau_{q_3^1}), (q_4^1, \tau_{q_4^1})\}$, where $\tau^1 = \{\neg x_2^1 x_4^1, \neg x_1^1(x_2^1 x_3^1 + \neg x_2^1 \neg x_4^1), x_1^1(x_2^1 x_3^1 + \neg x_2^1 \neg x_4^1), x_2^1 \neg x_3^1\}$.

It should be noted that to compute S^1 both functions y^2 and y^3 are needed. Considering only y^2 for the derivation of S^1 would be flawed due to the fact that two states in Q^1 that seem to be equivalent at output y^2 may possibly be distinguished at output y^3. Essentially the partition induced by both y_2 and y_3 is a refinement of the partition induced by y_2 only.

For S^2 derivation, functional decomposition on y^3 should be performed as is illustrated in Figure 3.2b.

Given an iterative combinational circuit $\mathcal{C}_C$, the state identification procedure for computing $S^0, \ldots, S^T$ is outlined in Algorithm 1. In line 1, S^0 and S^T are singleton sets as Q^0 has a single initial state q_1^0 and Q^T has a single don't-care destination state q_*^T. Moreover, the transition conditions to q_1^0 and q_*^T are tautologies. In lines 2-8, S^t for $t = 1, \ldots, T-1$ is computed through functional decomposition in line 7 on the hyperfunction encoded in line 4. Procedure *HyperEncode* encodes the output functions $Y^{t+1}, ..., Y^T$ into a single-output function h using the set A of fresh

doi:10.6342/NTU202001058



(a) Functional decomposition for S^1 derivation.

(b) Functional decomposition for S^2 derivation.

Figure 3.2: State identification.

new variables $\alpha_1, \ldots, \alpha_k$ for $k = \lceil \log_2(|Y^{t+1}| + \cdots + |Y^T|) \rceil$. Procedure *Decompose* performs functional decomposition on the hyperfunction h and extract the cut set and corresponding transition conditions.

Algorithm 1 StateIdentify

Input: $\mathcal{C}_C$ with inputs $X^1, ..., X^T$ and outputs $Y^1, ..., Y^T$
Output: $\{S^0, S^1, ..., S^T\}$
1: $S^0 := \{(q_1^0, 1)\}$; $S^T := \{(q_*^T, 1)\}$;
2: **for** $t = 1, ..., T-1$ **do**
3: $k := \lceil \log_2(|Y^{t+1}| + \cdots + |Y^T|) \rceil$;
4: $h := HyperEncode(Y^{t+1} \cup ... \cup Y^T, A = \{\alpha_1, \ldots, \alpha_k\})$;
5: $X_\lambda := X^1 \cup ... \cup X^t$;
6: $X_\mu := X^{t+1} \cup ... \cup X^T \cup A$;
7: $S^t := Decompose(h, X_\lambda, X_\mu)$;
8: **end for**
9: **return** $\{S^0, S^1, ..., S^T\}$;

3.2.2 Transition Reconstruction

With the sets $S^0, \ldots, S^T$ of state and transition condition pairs being obtained, the next step is to determine the transitions among the states and construct the state transition graph.

doi:10.6342/NTU202001058

Given an iterative combinational circuit $\mathcal{C}_C$, and the sets $S^0, \ldots, S^T$ as input, Algorithm 2 computes, for every pair (q_i^{t-1}, q_j^t) of states in adjacent two time-frames, the input condition and output response under the transition from q_i^{t-1} to q_j^t. Essentially, the input transition condition can be characterized by the QBF

$$\varphi_{i,j}^t = \exists X^1, \ldots, X^{t-1}.\tau_{q_i^{t-1}} \wedge \tau_{q_j^t} \tag{3.1}$$

and the output transition response can be characterized by the set of QBFs

$$\psi_{i,k}^t = \exists X^1, \ldots, X^{t-1}.\tau_{q_i^{t-1}} \wedge y_k^t \tag{3.2}$$

for $y_k^t \in Y^t$. In line 5, the procedure *TransitionTuple* returns the four-tuple $(q_i^{t-1}, q_j^t, \varphi_{i,j}^t, \{\psi_{i,k}^t \mid y_k^t \in Y^t\})$. The algorithm returns the collected four-tuples R for all state transitions. According to R, one can construct an FSM.

Algorithm 2 TransitionReconstruct

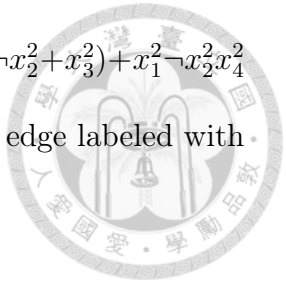
Input: $\mathcal{C}_C$, $\{S^0, \ldots, S^T\}$
Output: transition four-tuples R

```
1: R := ∅;
2: for t = 1, ..., T do
3:     foreach (q_i^{t-1}, τ_{q_i^{t-1}}) ∈ S^{t-1} do
4:         foreach (q_j^t, τ_{q_j^t}) ∈ S^t do
5:             R := R ∪ TransitionTuple(τ_{q_i^{t-1}}, τ_{q_j^t}, Y^t);
6:         end for
7:     end for
8: end for
9: return R;
```

Example 3.2 To illustrate, we derive the input condition for the transition from q_1^1 to q_1^2 shown on Figure 3.3a, where $\tau_{q_1^1} = \neg x_2^1 x_4^1$, $\tau_{q_1^2} = \neg x_2^1 x_4^1 \cdot (\neg x_1^2(\neg x_2^2 + x_3^2) + x_1^2 \neg x_2^2 x_4^2) + \neg x_1^1(x_2^1 x_3^1 + \neg x_2^1 \neg x_4^1) \cdot \neg x_2^2 x_4^2$. $y^2 = (\neg x_2^1 x_4^1 x_1^2 + \neg x_1^1(\neg x_2^1 \neg x_4^1 + x_2^1 x_3^1)) \cdot (x_2^2 + \neg x_4^2) + x_1^1(x_2^1 + \neg x_4^1) + \neg x_1^1 x_2^1 \neg x_3^1$. The input transition condition and the output

doi:10.6342/NTU202001058

transition response can be derived by: $\varphi^2_{1,1} = \exists X^1.\tau_{q_1^1} \wedge \tau_{q_1^2} = \neg x_1^2(\neg x_2^2 + x_3^2) + x_1^2 \neg x_2^2 x_4^2$ and $\psi_1^2 = \exists X^1.\tau_{q_1^1} \wedge y^2 = x_1^2(x_2^2 + \neg x_4^2)$, which corresponds to the edge labeled with "00--/0, 011-/0, 10-1/0" between q_1^1 and q_1^2 in Figure 3.3a.

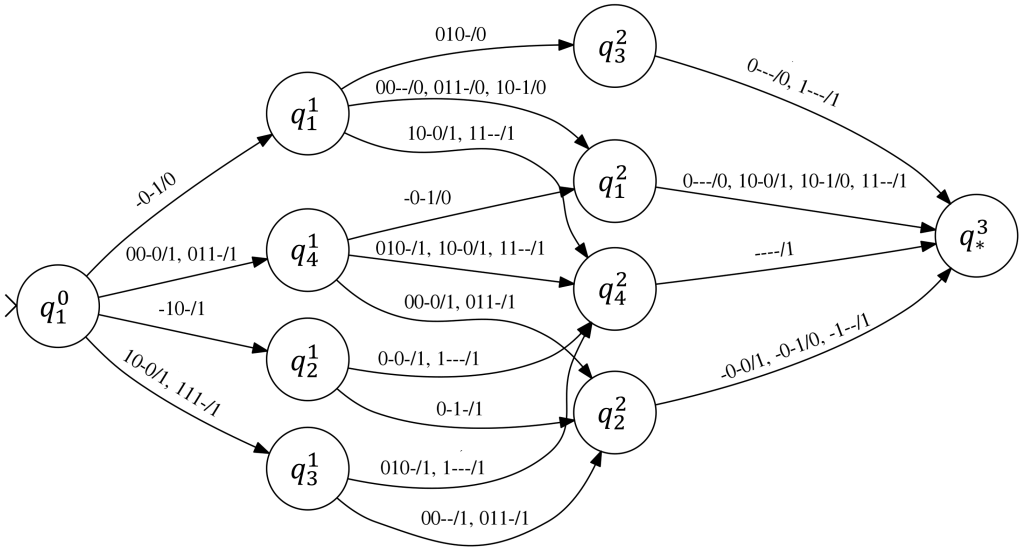


3.2.3 State Minimization

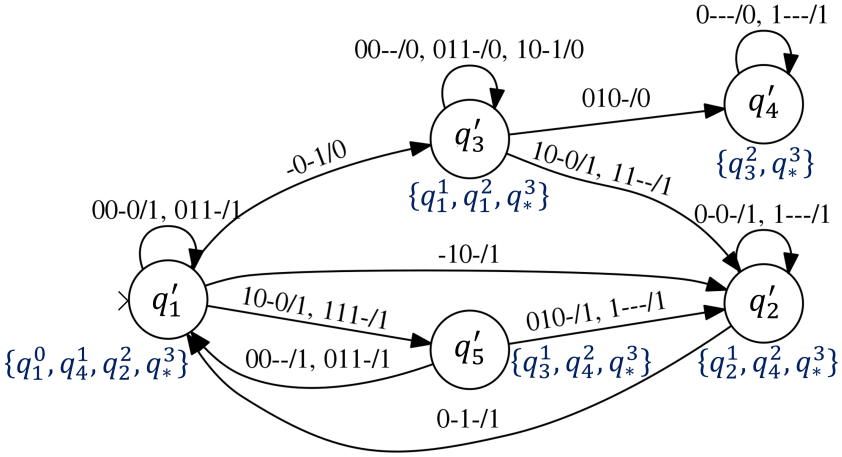
Notice that although by functional decomposition we guarantee that $|S^t|$ is minimum, the FSM constructed from R may not be state minimum. It is because equivalent states in different time-frames are not yet considered. In the FSM derived from time-frame folding, there is a unique initial state q_1^0 and final don't-care state q_*^T. As the FSM is incompletely specified at state q_*^T, the flexibility provides room for state minimization. In our implementation, we apply the SAT-based exact minimization algorithm `MeMin` [1] for FSM simplification.

Example 3.3 The FSM in Figure 3.3a can be minimized to that in Figure 3.3b. The number of states reduces from 10 (including the unspecified state q_*) to 5. In Figure 3.3b, each state is annotated with its compatible states in Figure 3.3a. In each time-frame except for the last, the states being identified are minimized such that none of them can be merged into the same state. For instance, the states reached at the first time-frame q_1^1, q_2^1, q_3^1 and q_4^1 in Figure 3.3a correspond to different states q_3', q_2', q_5' and q_1', respectively, in Figure 3.3b. Note that the minimized FSM in Figure 3.3b would not necessarily be equivalent to that of the original sequential circuit `s27` in Figure 3.3c, and more details are discussed in Subsection 5.1.1.

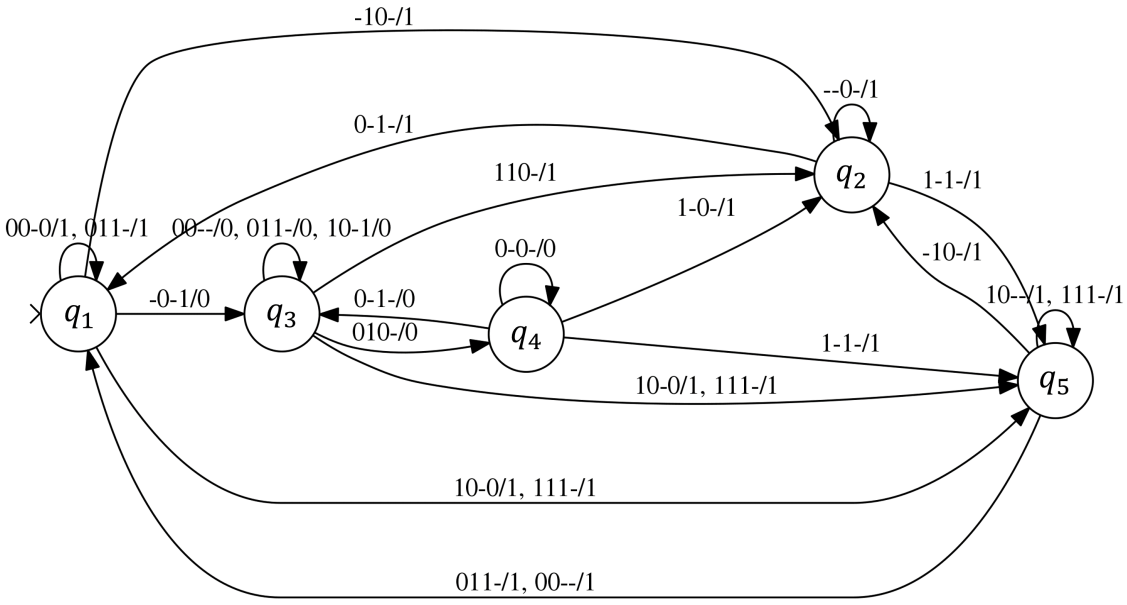
doi:10.6342/NTU202001058



(a) FSM from folding 3 time-frames (before state minimization).



(b) FSM from folding 3 time-frames (after state minimization).



(c) FSM of original `s27` circuit.

Figure 3.3: State transition graphs of FSMs.

doi:10.6342/NTU202001058

3.2.4 State Encoding

To transform an FSM into a sequential circuit, a final state-encoding step has to be performed. Let Q be the state set of the FSM. In our implementation, we try two different encoding schemes: 1) natural encoding, which uses $\lceil \log_2 |Q| \rceil$ bits, and 2) one-hot encoding, which uses $|Q|$ bits, each of which represents a state in Q.

3.3 Implementation Issues

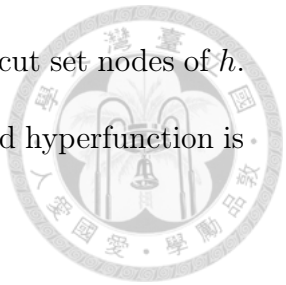
To improve state identification, we make two modifications to the *StateIdentify* algorithm:

- Reverse-chronological order enumeration: The index t in the for-loop in line 2 enumerates from 1 to $T-1$. As t increases, the number $|Y^t \cup ... \cup Y^T|$ of functions that have to be encoded decreases. Also there is a huge overlap of functions to be encoded at two consecutive time-frames t and $t+1$, which is $\{Y^{t+1}, ..., Y^T\}$. As a result, by reversing the enumeration order for t from $T-1$ to 1, the hyperfunction h can be built incrementally by adding Y^t to h one at a time in each iteration.

- Re-encoding hyperfunction: Now that the state and transition condition pairs identified at each time-frame are constructed in a reverse-chronological order, after we obtain S^t by decomposing the hyperfunction h built at time frame $t+1$, the variable in X^{t+1} is no longer relevant in deciding partition of $[\![X^1 \cup ... \cup X^t]\!]$. Hence, we can re-encode h into a more compact representation with less variables to reduce the circuit size. Essentially the variables X^{t+1} in h can be replaced with

doi:10.6342/NTU202001058

a new set of variables of size $\lceil \log_2 |S^t| \rceil$ in a way preserving the cut set nodes of h. Therefore h can be represented more compactly. The re-encoded hyperfunction is then be passed down to the next iteration.



doi:10.6342/NTU202001058

Chapter 4

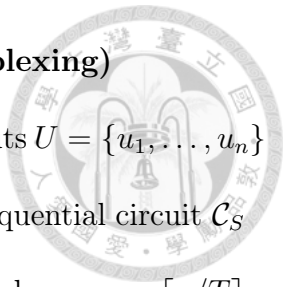
Circuit Folding for Time Multiplexing

In this chapter, we further extend the concept of "folding" for general combinational circuits, not exclusively for the iterative ones as described in Chapter 3, to achieve time multiplexing. In addition to the functional folding method, which exploits the time-frame folding technique, we also introduce the structural method, which only requires a traversal through the circuits for the intended objective. The chapter is organized as follows. The problem of time multiplexing is formulated in Section 4.1. Our algorithmic solutions are then presented in Sections 4.2 and 4.3.

4.1 Problem Formulation

The problem of *circuit folding for time multiplexing* can be stated as follows.

doi:10.6342/NTU202001058



Problem Statement 4.1 (Circuit Folding for Time Multiplexing) Given a folding number T and a combinational circuit $\mathcal{C}_C$ with inputs $U = \{u_1, \ldots, u_n\}$ and outputs $W = \{w_1, \ldots, w_{n'}\}$, we are asked to fold $\mathcal{C}_C$ into a sequential circuit $\mathcal{C}_S$ with inputs $X = \{x_1, \ldots, x_m\}$ and outputs $Y = \{y_1, \ldots, y_{m'}\}$, where $m = \lceil n/T \rceil$ and $m' \leq n'$, such that unfolding (expanding) $\mathcal{C}_S$ by T time-frames yields a combinational circuit $\mathcal{C}'_C$ with inputs $(X^1, \ldots, X^T)$ and outputs $(Y^1, \ldots, Y^T)$ that is functionally equivalent to $\mathcal{C}_C$ under some proper association of their inputs and outputs. That is, $\mathcal{C}_S$ achieves time multiplexing by taking T clock cycles, each taking m partial inputs, to execute the computation of $\mathcal{C}_C$.

In the sequel, we assume without loss of generality that n is divisible by T as one can always add dummy inputs (with no fanouts) to $\mathcal{C}_C$ to satisfy the divisibility.

We present two methods, structural circuit folding and functional circuit folding, for time multiplexing as follows.

4.2 Structural Circuit Folding

To find the sequential circuit $\mathcal{C}_S$ of the circuit folding problem of Section 4.1, let the inputs U of the given combinational circuit $\mathcal{C}_C$ be divided into T groups: $X^1 = \{u_1, \ldots, u_m\}$, $\ldots$, $X^T = \{u_{(T-1)\times m}, \ldots, u_n\}$. We then traverse the logic gates of $\mathcal{C}_C$ in a topological order by T iterations. At iteration t, for $t = 1, \ldots, T$, a topological traversal is initiated at the inputs X^t. A gate will be visited if and only if all of its fanins have been visited. On a visit to a gate in $\mathcal{C}_C$, a corresponding gate will be duplicated in $\mathcal{C}_S$. If a primary output of $\mathcal{C}_C$ is visited, then it will be scheduled

doi:10.6342/NTU202001058

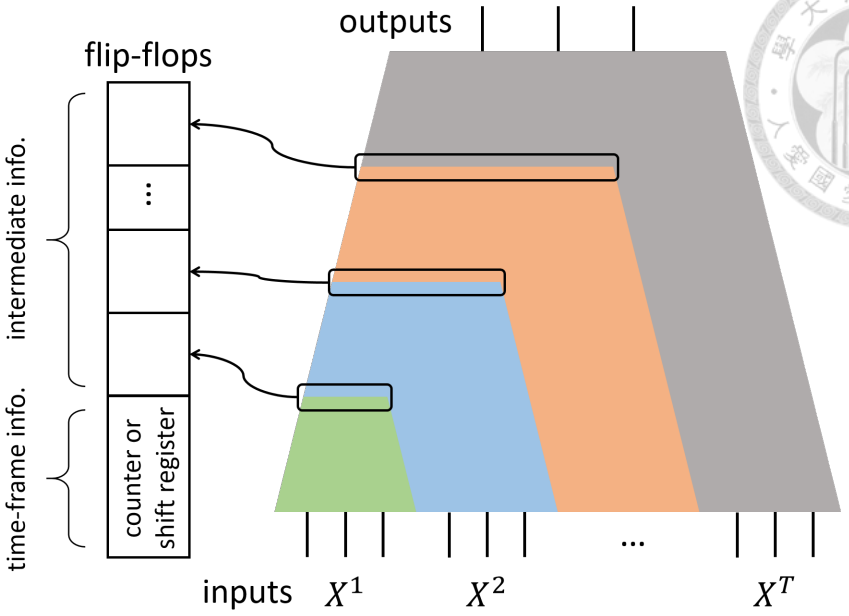


Figure 4.1: Illustration of structural circuit folding.

to output Y^t at time-frame t in $\mathcal{C}_S$. At the end of each iteration, the gates in the frontier of the traversal is collected, each of which has a newly introduced flip-flop in $\mathcal{C}_S$ to store its value. After T iterations, all the gates in $\mathcal{C}_C$ have been visited. Moreover, additional flip-flops are introduced to track the time-frame information, either with a $\lceil \log_2(T) \rceil$-bit counter using binary encoding or a T-bit shift register using one-hot encoding. The corresponding control logic is then added to select the correct output at each time-frame. Finally, we can obtain a sequential circuit $\mathcal{C}_S$ with inputs $X = \{x_1, \ldots, x_m\}$. The number of outputs of $\mathcal{C}_S$ is determined by the maximum number of outputs being scheduled in a time-frame among the T time-frames. Figure 4.1 illustrates the iterative-layering procedure of structural circuit folding. Different colors in the figure indicate the gate traversal at different time-frames. The frontier of each traversal is circled by a rounded rectangle and their signals are stored in the flip-flops, serving as the pseudo inputs to the circuit traversed in the next iteration.

doi:10.6342/NTU202001058

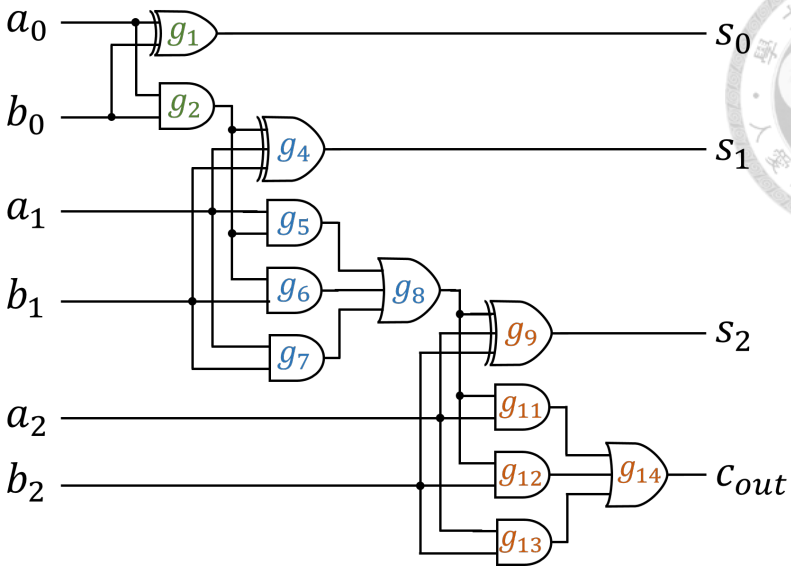


Figure 4.2: Example of 3-bit adder (`3-adder`) circuit under folding.

Example 4.1 To illustrate the procedure of structural circuit folding, we take the 3-bit adder in Figure 4.2 as an example. The adder has inputs $U = A \cup B$ and outputs $W = \{s_0, s_1, s_2, c_{out}\}$, where $A = \{a_0, a_1, a_2\}$ and $B = \{b_0, b_1, b_2\}$ are the 2 input 3-bit numbers, with a_i and b_i being the i^{th} bits of A and B, respectively, s_i the i^{th} summation bit, and c_{out} the carry-out bit. The inputs are grouped as $X^1 = \{a_0, b_0\}$, ..., $X^3 = \{a_2, b_2\}$. The gates in Figure 4.2 marked in green, blue, and orange correspond to the gates visited at the first, second, and third iteration, respectively. A total of 5 flip-flops are introduced, 2 for storing the intermediate information of g_2 and g_8, which are essentially the carry bits of the first two iterations, and 3 for storing the time-frame information as a shift register. The number of outputs of $\mathcal{C}_S$ is determined by $|Y^3| = 2$. The outputs are scheduled as follows: $Y^1 = \{s_0, null\}$, $Y^2 = \{s_1, null\}$, and $Y^3 = \{s_2, c_{out}\}$, where *null* denotes a dummy output. With the control logic being added for selecting the correct output at each time-frame, $\mathcal{C}_S$ can be synthesized to a circuit with 2 inputs, 2 outputs, 5 flip-flops, and 23 AIG nodes (or 8 6-input LUTs) [7].

doi:10.6342/NTU202001058

Although the structural circuit folding method is efficient and scalable to large circuits, the constructed sequential circuit $\mathcal{C}_S$ can be sub-optimal. Taking `3-adder` of Figure 4.2 for example, we know that ultimately $\mathcal{C}_S$ can be implemented with an 1-bit carry-save adder, consisting of only 1 input, 2 outputs, 1 flip-flop, and 7 AIG nodes (for a full adder implementation). It motivates the functional circuit folding approach as we present next.

4.3 Functional Circuit Folding

We exploit the time-frame folding (TFF) technique in Chapter 3 to the time multiplexing problem. Note that the original TFF cannot be applied directly because it assumes the given combinational circuit under folding is in an iterative form. However, time multiplexing must work for general combinational circuits not necessarily iterative ones. Below we detail the functional circuit folding method.

As shown in Figure 4.3, the functional circuit folding algorithm consists of three main computation steps: 1) pin scheduling, 2) FSM construction via time-frame folding, 3) FSM minimization, and 3) FSM encoding, to be presented in the following subsections.

4.3.1 Pin Scheduling and Iterative Circuit Conversion

Given a folding number T and a combinational circuit $\mathcal{C}_C$ with inputs $U = \{u_1, \ldots, u_n\}$ and outputs $W = \{w_1, \ldots, w_{n'}\}$, the pin scheduling procedure permutes the inputs and outputs (and possibly adds dummy inputs and outputs) to convert

doi:10.6342/NTU202001058

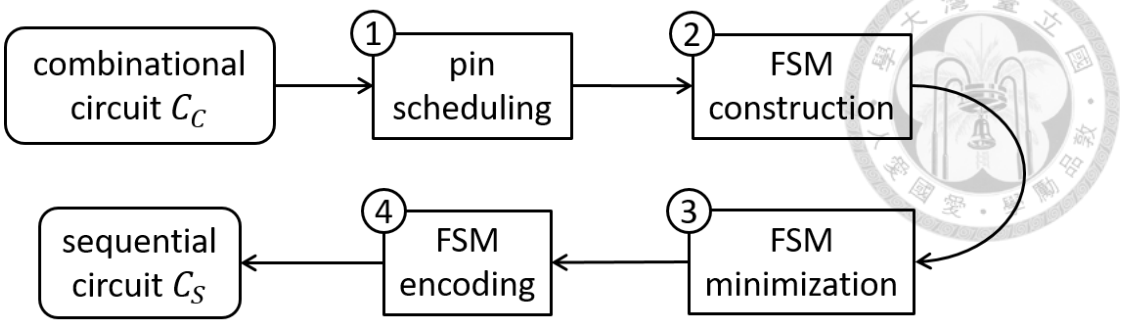


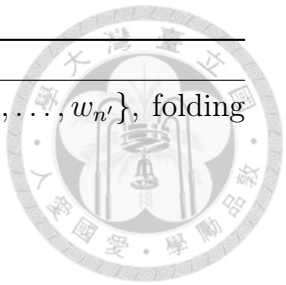
Figure 4.3: Computation flow of functional circuit folding.

$\mathcal{C}_C$ into a *virtual* T-iterative combinational circuit $\mathcal{C}'_C$ with inputs $X^1, \ldots, X^T$ for $X^t = \{x_1^t, \ldots, x_m^t\}$ and outputs $Y^1 \ldots Y^T$ for $Y^t = \{y_1^t, \ldots, y_{m'}^t\}$, where $m = \lceil n/T \rceil$ and $m' \leq n'$. The circuit after scheduling must satisfy the property that every primary output $w_i \in W$ is scheduled at some iteration t while its input supports are scheduled in iterations $t' \leq t$.

Algorithm 3 shows a heuristic scheduling procedure of outputs $W = \{w_1, \ldots, w_{n'}\}$ with respect to a folding number T. In line 1, the number m of inputs in one circuit iteration is calculated. In line 2, the set of outputs W is sorted according to their support sizes in an ascending order. In line 3, the sets $U_{sup}, Y^1, \ldots, Y^T$ are initialized to be empty. In lines 4-8, the loop goes over each output w_i to determine its iteration. In line 5, the support set of w_i is added to U_{sup}. In line 6, the earliest available iteration t for w_i is calculated. In line 7, w_i is assigned to Y^t. Finally, the output schedule is returned in line 9. Note that to make the number of outputs scheduled at each iteration identical, null (dummy) outputs are inserted to $Y^1, \ldots, Y^T$. In our implementation, we also try to minimize the number of outputs by prolonging some of the scheduled outputs.

With the outputs being scheduled, the inputs $W = \{w_1, \ldots, w_{n'}\}$ can be sched-

doi:10.6342/NTU202001058



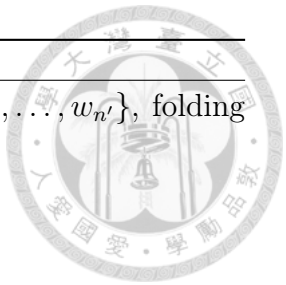
Algorithm 3 OutputSchedule

Input: $\mathcal{C}_C$ with inputs $U = \{u_1, \ldots, u_n\}$ and outputs $W = \{w_1, \ldots, w_{n'}\}$, folding number T
Output: output schedule $Y^1, \ldots, Y^T$
1: $m := n/T$;
2: $SortAscend(W)$;
3: $U_{sup}, Y^1, \ldots, Y^T := \emptyset$;
4: **foreach** w_i in W **do**
5: $\quad U_{sup} := U_{sup} \cup Support(w_i)$;
6: $\quad t := \lceil |U_{sup}|/m \rceil$;
7: $\quad Y^t := Y^t \cup \{w_i\}$;
8: **end for**
9: **return** $(Y^1, \ldots, Y^T)$;

uled accordingly as outlined in Algorithm 4. Let X_{que} be a queue to store the ordered inputs. In line 1, X_{que} is initialized as an empty queue. In lines 2-6, the loop iterates through each scheduled outputs Y^t to fill in the queue. In line 3, the supports X_{sup} of Y^t that have not yet been scheduled during the previous iterations are collected in queue X_{sup}. In line 4, an optional optimization step is performed to reorder X_{sup}. Since the FSM construction algorithm in the later step relies on BDD-based operations, a smaller BDD size of $\mathcal{C}'_C$ would help to reduce the execution time. Therefore, BDD variable reordering with symmetric sifting [26] technique is applied to X_{sup} to minimize the BDD size of outputs Y^t of $\mathcal{C}_C$. In line 5, X_{sup} is pushed into the queue X_{que}. In line 7, X_{que} is evenly divided into T groups $X^1, \ldots, X^T$, which are finally returned in line 8.

Example 4.2 Consider the `3-adder` example in Figure 4.2. After pin scheduling, we have outputs $Y^1 = \{s_0, null\}$, $Y^2 = \{s_1, null\}$, $Y^3 = \{s_2, c_{out}\}$, and inputs $X^1 = \{a_0, b_0\}$, $X^2 = \{a_1, b_1\}$, $X^3 = \{a_2, b_2\}$. Note that the null (dummy) outputs are inserted to make the number of outputs scheduled at each iteration identical.

doi:10.6342/NTU202001058



Algorithm 4 InputSchedule

Input: $\mathcal{C}_C$ with inputs $U = \{u_1, \ldots, u_n\}$ and outputs $W = \{w_1, \ldots, w_{n'}\}$, folding number T, and output schedule $Y^1, \ldots, Y^T$
Output: input schedule $X^1, \ldots, X^T$

```
1: X_que := ∅;
2: for t = 1, ..., T do
3:     X_sup := Support(Y^t) \ X_que;
4:     X_reord := BddSymSift(C_C, t, X_sup);
5:     X_que := Append(X_que, X_reord);
6: end for
7: (X^1, ..., X^T) := Split(X_que, T);
8: return (X^1, ..., X^T);
```

4.3.2 FSM Construction via Time-Frame Folding

Given an T-iterative combinational circuit $\mathcal{C}'_C$ with inputs $X^1, \ldots, X^T$ for $X^t = \{x_1^t, \ldots, x_m^t\}$ and outputs $Y^1, \ldots, Y^T$ for $Y^t = \{y_1^t, \ldots, y_{m'}^t\}$, the TFF algorithm in Chapter 3 can be applied to construct an FSM with inputs $X = \{x_1, \ldots, x_m\}$ and outputs $Y = \{y_1, \ldots, y_{m'}\}$, which has the same input-output behavior as $\mathcal{C}'_C$ within the T bounded time-frames. The algorithm relies on BDD-based functional decomposition to identify the internal states and construct the transitions between states according to the identified state information. Some minor modifications to the TFF algorithm are needed as we discuss below. In Chapter 3, the iterative circuit being folded or transformed is fully-specified, that is, there are no null output functions. Because null functions do not provide any additional information for state partitioning, they can simply be discarded from $Y^{t+1}, \ldots, Y^T$ or be treated as constant functions during the encoding stage of state identification. Similarly, when determining the output response of a state at time-frame t, if there is a null output scheduled at that time-frame, then its corresponding slot should remain unspecified. Following the notation of Chapter 3, $Q^t = \{q_1^t, \ldots, q_k^t\}$ is used to denote the set of

doi:10.6342/NTU202001058

states identified at time-frame t.

4.3.3 FSM Minimization

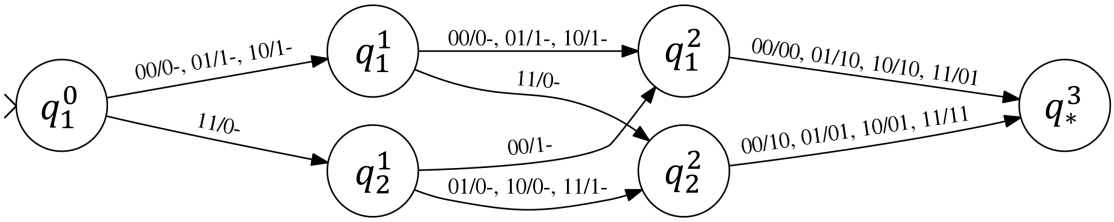
In the derived FSM, there is a unique initial state s_1^0 and a don't-care destination state s_*^T inserted by the TFF algorithm, along with some null (dummy) outputs at several states. As the FSM is incompletely specified, the flexibility leaves room for state minimization. Again, we adopt the SAT-based exact minimization algorithm `MeMin` [1] for FSM simplification as in Chapter 3.

Example 4.3 The state diagram in Figure 4.4a, where the mark ">" indicates the initial state, is obtained by folding the `3-adder` circuit by 3 time-frames with the functional circuit folding algorithm. It can be further minimized to that in Figure 4.4b. The number of states reduces from 6 (including the don't-care state s_*^3) to 2. In Figure 4.4b, each state is annotated with its compatible states in Figure 4.4a. We can observe that the minimized FSM is essentially a carry-save adder, where s'_0 and s'_1 corresponds to the state with carry-bit of value 0 and 1, respectively.

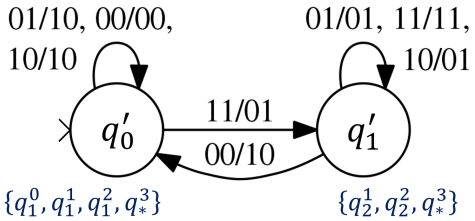
4.3.4 FSM Encoding

The step is identical to that described in Subsection 3.2.4. Two encoding methods: 1) natural binary encoding and 2) one-hot encoding are applied to convert the FSM into a sequential circuit.

doi:10.6342/NTU202001058



(a) FSM before state minimization.



(b) FSM after state minimization.

Figure 4.4: FSM by functional circuit folding of `3-adder`.

doi:10.6342/NTU202001058

Chapter 5

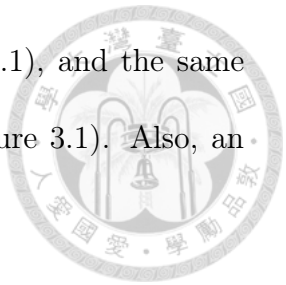
Experiments

In this chapter, we evaluated our proposed methods on circuits selected or converted from several sets of benchmarks, including ISCAS, ITC, MCNC(LGSynth), LEKO/LEKU, Adder, and EPFL benchmarks. The proposed algorithmic methods were implemented in C++ language within the `ABC` system [7], which utilized `CUDD` [31] as the underlying BDD package. Moreover, an open source package `MeMin` [1] was used for state minimization. All the experiments were conducted on a Linux server with Intel(R) Core(TM) i7-8700 3.20GHz CPU and 32GB RAM.

5.1 Time-frame Folding

TFF algorithm was evaluated with respect to three sets of benchmark circuits. Two were obtained from unfolded and simplified ISCAS and ITC circuits, and one was obtained from QBF solving of adaptive homing sequences [35]. A timeout limit of

doi:10.6342/NTU202001058



300 seconds is imposed on `timefold` (steps 1 and 2 in Figure 3.1), and the same limit is imposed on `MeMin` for state minimization (step 3 in Figure 3.1). Also, an expansion limit of 5000 time-frames was imposed.

The results on ISCAS and ITC benchmarks are shown in Table 5.1, where Columns 2-5 list the numbers of primary inputs, primary outputs, latches, and AIG nodes, respectively, after optimization of the original sequential circuits, Column 6 lists the maximum time-frames that can be expanded and folded back within the timeout limit, Columns 7 and 8 list the numbers of states of the folded circuit before and after state minimization, respectively. For an entry in the table containing two values separated by "/", it indicates that `MeMin` reached its timeout limit before `timefold` reached its maximum number of time-frames. The value on the left of "/" shows the data that both `timefold` and `MeMin` are executed successfully, while the value on the right shows the data that only `timefold` can be done within the timeout limit. Circuits `b01` and `b02` reached the 5000 time-frame limit and are marked with the "*" sign.

From the table, the numbers of foldable time-frames within 300 seconds vary to some extent, roughly proportional to the growth rate of the number of states. On the other hand, the performance of `MeMin` exhibited somewhat non-robustness. For example, for `s382` expanded with 51 time-frames, the 11983 states can be successfully minized to 1367 states within 300 seconds; in contrast, for `s713` expanded with 4 time-frames, the 75 states cannot be minimized within 300 seconds. For the homing sequence benchmarks, the results are shown in Table 5.2. As the depths of the adaptive homing strategies are not large, our method successfully generates all

doi:10.6342/NTU202001058

sequential circuits.

Table 5.1: Results of TFF on ISCAS and ITC benchmarks.

circuit	#PI	#PO	#FF	#gate	#frm	#state	#m-state
b01	2	2	5	38	5000*	22493	18
b02	1	1	4	16	5000*	9997	8
b03	4	4	21	55	88	24968	631
b04	11	8	66	333	4/5	132/77195	130/-
b05	1	36	34	405	621	37804	69
b06	2	6	8	26	367	4367	13
b07	1	8	39	320	520	37599	83
b08	9	4	21	122	72	29003	798
b09	1	1	28	120	24/32	10241/96299	3795/-
b10	11	6	16	151	16/24	3248/10746	602/-
b11	7	6	30	469	15/21	2542/29458	676/-
b12	5	6	119	910	107	10317	1104
b13	10	10	45	168	117/158	10276/211252	139/-
b14	32	54	215	3689	2	3	2
b15	36	70	415	6587	6	11	8
b17	37	97	573	7648	7/11	103/13826	93/-
b18	36	23	3320	0	92	17444	382
b20	32	22	429	7956	2	3	1
b21	32	22	429	8067	2	3	1
b22	32	22	611	12339	2	3	1
s27	4	1	3	8	189	940	5
s208.1	10	1	8	48	183	12685	129
s298	3	6	14	70	55	5841	135
s344	9	11	15	91	5/43	1262/39618	863/-
s349	9	11	15	91	5/44	1262/40634	863/-
s382	3	6	21	92	51	11983	1367
s386	7	7	6	81	115	1458	13
s400	3	6	21	92	51	11983	1367
s420.1	18	1	16	101	187	13201	129
s444	3	6	21	95	51	11983	1367
s499	1	22	22	118	416	8922	22
s510	19	7	6	204	45/88	967/2984	44/-
s526	3	6	21	88	51	12021	1370
s641	35	24	14	100	2/4	3/75	2/-
s713	35	23	14	100	2/4	3/75	2/-
s820	18	19	5	200	62	1336	24
s832	18	19	5	215	67	1456	24
s838.1	34	1	32	214	189	13459	129
s938	34	1	32	214	188	13330	129
s953	16	23	29	282	9/21	270/6146	111/-
s967	16	23	29	285	9/20	270/6096	111/-
s991	65	17	19	331	1/2	2/6	1/-
s1196	14	14	18	367	1/3	2/1934	1/-
s1238	14	14	18	394	1/3	2/1934	1/-
s1269	18	10	37	413	1/2	2/4340	1/-
s1423	17	5	73	435	6/8	498/15698	396/-
s1488	8	19	6	472	77	3150	48
s1494	8	19	6	484	82	3390	48
s1512	29	21	57	342	5/9	32/2019	24/-
s3271	26	14	115	836	11	185	154
s3330	40	73	65	557	1	2	1
s3384	43	26	183	1006	5	13	12
s4863	49	16	81	789	1/2	2/1542	1/-
s5378	35	49	127	736	1	2	1
s6669	83	55	231	2226	0	-	-
s9234.1	36	39	129	759	0/2	-/10	-/-
s13207	31	121	193	547	11/13	8034/27394	8033/-
s15850	14	87	128	375	769/770	776/777	11/-
s35932	35	320	1472	7345	5	90	53
s38417	28	106	1345	7179	2/3	6/114	5/-
s38584	12	278	784	4456	5/6	162/754	141/-

doi:10.6342/NTU202001058

Table 5.2: Results of time-frame folding on homing sequence benchmarks.

circuit	#PI	#PO	#gate	#frm	#state	#m-state
5s2i0_c	3	3	1	3	6	4
5s2i0_r	3	3	0	3	4	1
5s2i2_c	4	4	1	4	9	4
5s2i2_r	4	4	2	4	8	5
10s5i1_c	12	12	175	4	35	29
10s5i4_c	12	12	105	4	30	24

To better understand the relation among the number of states, the number of time-frames, and the runtime, circuits `b07`, `b18`, `s499`, `s386`, `s1494`, and `s15850` were selected for study. Figure 5.1 shows the relation between the number of states and the number of expanded time-frames. It can be observed that the number of states before minimization (solid lines) constantly increased with the number of time-frames, whereas the number of states after minimization (dotted lines) tended to saturate after a certain number of time-frames. This phenomenon is expectable as all inequivalent states should be distinguished eventually. On the other hand, Figure 5.2 shows the relation between the total runtime (in gray) of `timefold` and `MeMin` combined and the number of time-frames. The runtimes of `timefold` and `MeMin` are plotted as bar charts every 5 time-frames in blue and green, respectively. The positive correlation between the runtime and the number of expanded time-frame is expected.

5.1.1 Fixed Point after TFF

We verified the consistency between the constructed sequential circuits and their corresponding expanded iterative combinational circuits. In the cases of our experiments, we observed that the constructed sequential circuit tends to become sequentially equivalent to its original sequential circuit when the number of expanded

doi:10.6342/NTU202001058

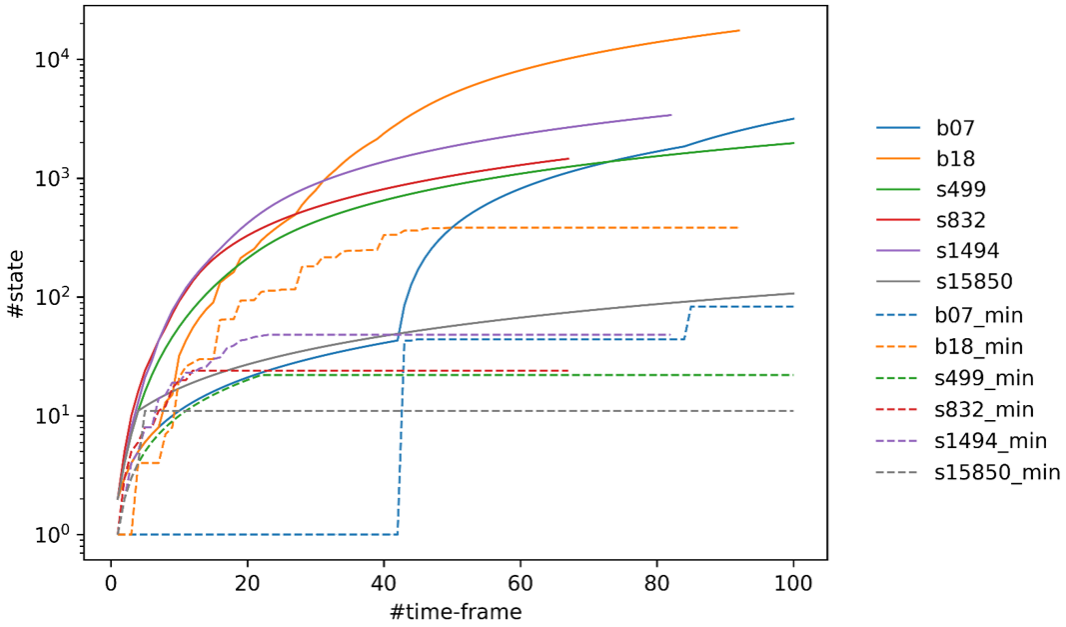
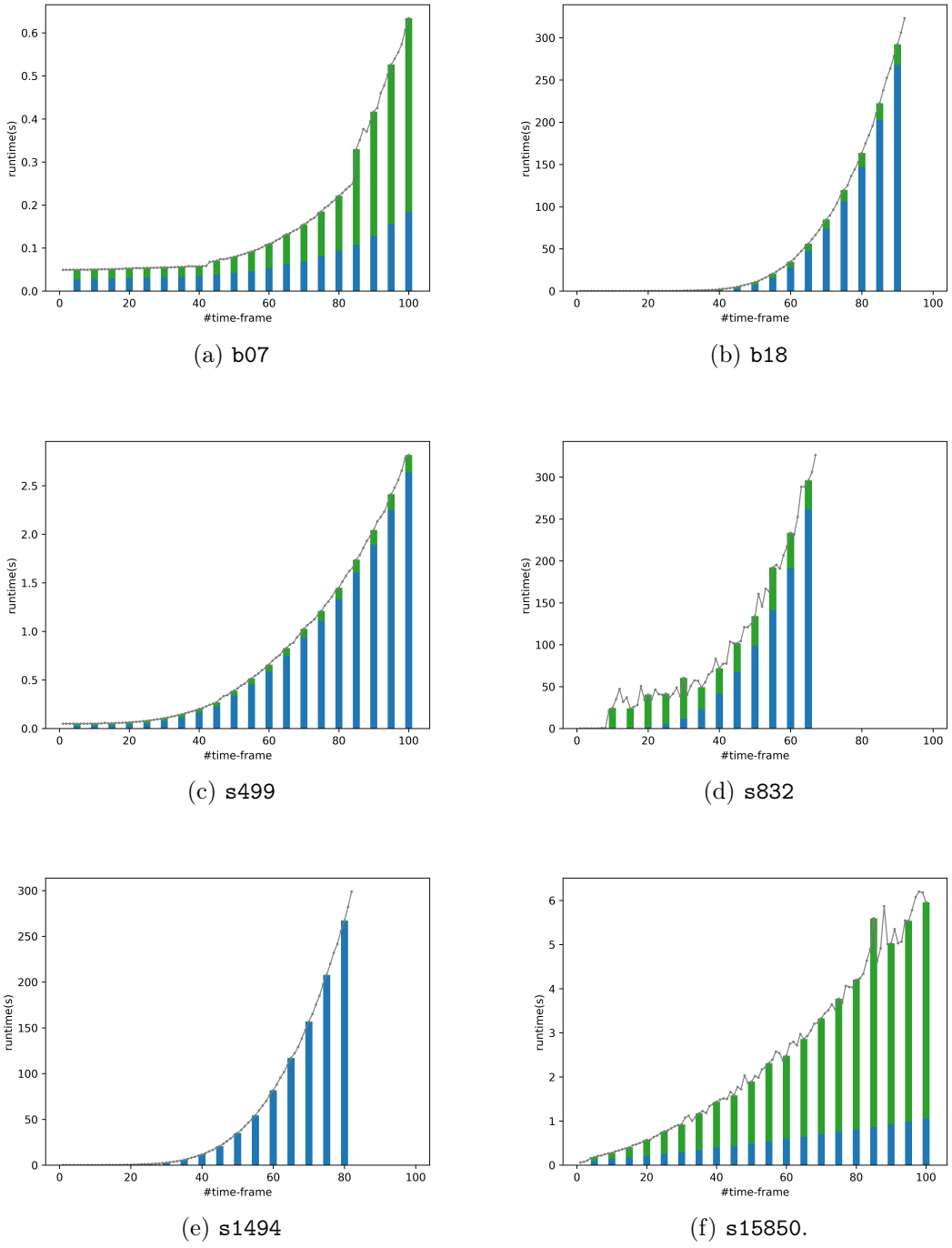


Figure 5.1: #state vs. #time-frame.

time-frames is sufficiently large. We call this phenomenon as a *fixed point*. However, the sequential equivalence may not happen immediately at the time-frame when the number of states starts to saturate. Let q_1 be the initial state of the state transition graph, $m_{i,j}$ be the length of the shortest path from state q_i to state q_j, and $n_{i,j}$ be the length of the shortest sequence distinguishing states q_i and q_j. Also let $\mathbf{m}$ be the maximum length among the shortest paths from the initial state to any other states, i.e., $\mathbf{m} = \max\{m_{1,j}\}$, for any $q_j \in Q, j \neq 1$; let $\mathbf{n}$ be the maximum length among the shortest sequences distinguishing any state pairs, i.e., $\mathbf{n} = \max\{n_{i,j}\}$, for any $q_i, q_j \in Q, i \neq j$. In fact, if the reachable state sets grow monotonically during time-frame expansion, then the fixed point is guaranteed by expanding the circuit no greater than $\mathbf{m} + \mathbf{n}$ time-frames.

Table 5.3 shows the time-frame numbers in columns 2-3 when the number of states starts to saturate and when the obtained circuit starts to become sequen-

doi:10.6342/NTU202001058



(a) b07 (b) b18

(c) s499 (d) s832

(e) s1494 (f) s15850.

Figure 5.2: Total runtime vs. #time-frame.

doi:10.6342/NTU202001058

tially equivalent to the original circuit. Note that not every considered circuit is listed in Table 5.3, because some of them are not able to reach these two conditions within their maximally expanded time frames. The table lists in columns 4-6 the information of the original sequential circuits and the expanded combinational circuits, which are expanded respectively by the number of time-frames that reaches fixed point. Also, columns 7-12 shows the numbers of flip-flops and gates of the folded sequential circuit under two different encoding schemes, and the corresponding reduction ratio on the numbers of gates compared to those of its corresponding time-frame expanded circuit. As generally observed, natural encoding can result in fewer flip-flops, but require more gates, while one-hot encoding can achieve better gate count reduction, but require more flip-flops.

Table 5.3: Results on folding with fixed point.

circuit	#frame		original		expn.	natural encoding			one-hot encoding		
	sat.	fp.	#FF	#gate	#gate	#FF	#gate	redu.	#FF	#gate	redu.
b01	9	9	5	38	52	5	104	-100.00%	18	52	0.00%
b02	6	10	4	16	4	3	16	-300.00%	8	16	-300.00%
b03	14	14	21	55	189	10	8947	-4633.86%	631	1848	-877.78%
b05	69	133	34	405	35173	7	52	99.85%	69	11	99.97%
b06	6	7	8	26	52	4	82	-57.69%	13	45	13.46%
b07	85	85	39	320	13822	7	75	99.46%	83	54	99.61%
b08	55	55	21	122	5538	10	3395	38.70%	798	1083	80.44%
b18	50	50	129	2178	33139	9	2516	92.41%	382	1068	96.78%
s27	3	5	3	8	29	3	23	20.69%	5	42	-44.83%
s298	20	23	14	70	838	8	1489	-77.68%	135	767	8.47%
s386	8	9	6	81	297	4	117	60.61	13	74	75.08
s499	22	23	22	118	1333	5	71	94.67%	22	86	93.55%
s820	12	13	5	200	1484	5	276	81.40%	24	1360	8.36%
s832	12	13	5	215	1390	5	248	82.16%	24	1245	10.43%
s1488	23	23	6	472	7422	6	492	93.37%	48	341	95.41%
s1494	23	23	6	484	7693	6	523	93.20%	48	334	95.66%
s15850	5	5	128	375	24	4	29	-20.83%	11	24	0.00%

5.1.2 Circuit Size Compaction

To verify that our proposed method indeed has the ability in circuit size compaction, we compared the sizes of the expanded combinational circuits to their folded sequential circuits in terms of AIG nodes. The ISCAS and ITC benchmark circuits selected

doi:10.6342/NTU202001058

for comparison are the ones that have reached the number of time-frames to observe sequential equivalence, and are expanded by that number of time-frames. Note that for time-frame folding, there is no need to expand more than that number of time-frames, since the folded sequential circuit will remain the same, while the expanded combinational circuit will continue to grow in size. Additionally, homing sequence benchmarks are also included for comparison. The results are plotted in Figure 5.3, where black data points correspond to ISCAS and ITC benchmark circuits, and the blue ones correspond to homing sequence benchmarks. Both natural and one-hot encoding schemes were applied, and the one resulted in a smaller circuit size was taken for comparison. The data points on the right of the gray dotted line correspond to the cases where the obtained sequential circuits are of size smaller than their combinational counterparts. We observed that larger circuits tend to benefit more from our method, as the combinational circuits with over 200 AIG nodes, when folded into sequential circuits, are all reduced significantly in size. Note that the upper-most (worst-case) point in Figure 5.3 is the circuit `b03` expanded with 14 time-frames. Although time-frame folding does not achieve compaction in this case, it is expected that, when more time-frames are to be expanded, the iterative combinational circuit size will keep growing while the folded sequential circuit size will remain the same.

5.2 Time Multiplexing via Circuit Folding

The proposed structural and functional methods were evaluated on 27 combinational circuits shown in Table 5.4, where columns 2-5 list the numbers of primary inputs,

doi:10.6342/NTU202001058

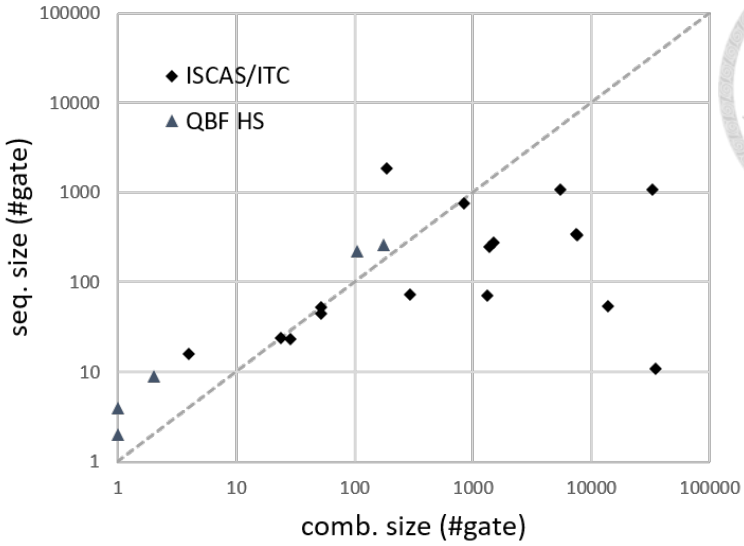


Figure 5.3: Circuit size after TFF.

primary outputs, AIG nodes, and 6-input LUTs, respectively, of the circuits after optimization. The circuits marked with "*" are simplified from the original circuits by extracting some primary outputs and keeping only the structural input support of those outputs.

Table 5.4: Benchmark statistics.

circuit	#PI	#PO	#gate	#LUT
64-adder	128	65	507	96
128-adder	256	129	844	244
128-parity	128	1	381	33
apex2	38	3	1448	581
arbiter*	256	1	361	102
b14_C	276	299	3890	1152
b15_C	484	519	6801	1966
b17_C*	380	3	1634	381
b20_C	521	512	8173	2221
b21_C	521	512	8250	2311
b22_C	766	757	12355	3375
bcb	26	39	1554	530
C7552	207	108	1485	340
des	256	245	3087	717
e64	65	65	244	114
g216	216	216	3982	648
g625	625	625	10625	2498
g1296	1296	1296	31447	5184
hyp	256	128	213158	45142
i2	201	1	208	63
i3	132	6	126	38
i4	192	6	186	42
i10	257	224	1586	507
max	512	130	2776	812
mem_ctrl	1204	1231	15908	5207
too_large	38	3	2642	1111
voter	1001	1	12400	1667

doi:10.6342/NTU202001058

5.2.1 Structural Folding on Large Circuits

We first evaluate the effectiveness of the structural method for time multiplexing by imposing the I/O pin count limitation to 200, according to some commercial FPGA specifications. In addition, a simple alternative to fold a circuit by T time-frames can be done by temporarily storing inputs of the first $T-1$ time-frames into flip-flops and defer computing all outputs at the last time-frame. Table 5.5 shows the results on folding 17 benchmark circuits with more than 200 pins under 5 different settings. Due to the space limitation, the table is split into 2 subtables. Columns 2-3 of each subtable list the number of time-frames each circuit should be folded and the number of inputs after folding, and the rest of the columns list the information of the folded circuit under different settings. The "simple" setting corresponds to the method by temporarily storing inputs described earlier, and the "structural" setting corresponds to the structural method presented in Section 4.2. In the settings annotated with "s", we applied pin scheduling procedure outlined in Subsection 4.3.1. In the settings annotated with "f", flip-flops were reused during folding to lower the flip-flop usage, under the condition that the value held by a flip-flop at current iteration is no longer needed in the computation of the following iterations. Under each setting, the 5 columns list the information of folded sequential circuit, including the number of outputs, flip-flops, AIG nodes, 6-input LUTs, and the LUT overhead incurred comparing to the original combinational circuit, respectively. The average LUT overhead is listed in the last row of Table 5.5. The experimental results indicate the ability of the structural method on meeting the I/O pin constraint[3]. The circuit

[3]Note that the number of output pins can be larger than 200. In that case, multiple clock cycles can be taken to produce the outputs.

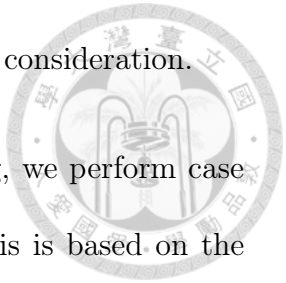
doi:10.6342/NTU202001058

size in terms of LUT usage before and after folding is plotted in Figure 5.4. The data points on the left of the gray dotted line correspond to the cases where the folded circuits are of size larger than the original combinational circuits. It can be observed that circuit folding would incur some LUT overhead in most cases.

The best overall results of structural folding were obtained by applying both the pin scheduling and flip-flop reuse procedures, and incurred an average of 20.07% LUT overhead, despite the fact that there are cases, `128-adder`, `hyp`, `i2`, with LUT savings. Notice that the LUT increase could not be a serious problem as the LUT resources are not as critical as the I/O pin bottleneck in FPGAs. The experiments show that the pin scheduling procedure can reduce the number of output pins of the folded circuit, and that flip-flop reuse procedure can lower the flip-flop usage. Therefore, the 2 methods combined can result in a folded circuit with lower complexity (less LUT usage). As all the experiments were done in less than a second, the results demonstrate the scalability of the structural method.

When applied to the 17 benchmark circuits in Table 5.5, the additional control circuitry to store the input signals of the simple method incurred an average 46.59% LUT overhead, which is 26.52% higher than the proposed structural-sf method. The number of flip-flops required for the simple method is larger or equal to the structural-sf method in all cases. The number of output pins after this simple folding remains the same as the number of primary outputs of the original combinational circuit, since all the outputs are scheduled to be computed at the last time-frame. In contrast, the structural method can achieve output pin reduction on 11 out of the 17 cases. In comparison, the structural-sf method is better than the simple method

doi:10.6342/NTU202001058



when taking the number of LUTs, flip-flops and output pins into consideration.

To study the potential of latency reduction by circuit folding, we perform case analysis on circuit `i10`, with 257 PIs and 224 POs. The analysis is based on the following assumptions: 1) Assume the maximum I/O transmission rate is 200 bits per I/O clock cycle. 2) Assume TDM ratio $r = 1$, i.e., the system clock cycle equals the I/O clock cycle, for the circuit without folding and the circuit with folding. 3) Assume the combinational logic of both circuits without and with folding can be computed in one I/O clock cycle. With structural circuit folding, `i10` would be folded by two time-frames into a sequential circuit with 129 inputs and 180 outputs as shown in Table 5.5, with 44 outputs scheduled in the first time-frame and 180 scheduled at the second time-frame. The overall execution requires three system (also I/O) clock cycles: the first cycle transmits 129 inputs, second cycle 129 inputs and 44 outputs, and third cycle 180 outputs. In contrast, without circuit folding, the execution of `i10` requires a total of four I/O clock cycles: the first cycle transmits 200 inputs, second cycle 57 inputs, third cycle 200 outputs, and fourth cycle 24 outputs. Effectively, circuit folding may achieve 25% I/O clock cycle reduction. In fact, TDM aims at increasing the effective I/O pins of FPGA by slowing down the system clock to increase I/O transmissions during a system clock period, while our circuit folding can directly decrease the required number of pins of a logic circuit. The TDM and circuit folding methods are orthogonal, and can be combined to alleviate the FPGA I/O bottleneck issue.

doi:10.6342/NTU202001058

Table 5.5: Results of structural circuit folding.

circuit	#frm	#in	simple					structural					structural-s				
			#out	#FF	#gate	#LUT	overhead	#out	#FF	#gate	#LUT	overhead	#out	#FF	#gate	#LUT	overhead
128-adder	2	128	129	129	1425	410	68.03%	65	2	641	195	-20.08%	65	2	641	195	-20.08%
b14_C	2	138	299	139	4451	1313	13.98%	262	453	5213	1540	33.68%	189	347	4973	1427	23.87%
b15_C	3	162	519	327	8211	2367	20.40%	274	561	8928	2473	25.79%	343	838	9276	2655	35.05%
b20_C	3	174	512	351	9617	2663	19.90%	424	734	10654	2964	33.45%	248	536	10000	2619	17.92%
b21_C	3	174	512	351	9661	2739	18.52%	424	726	10497	2952	27.74%	248	548	10019	2703	16.96%
b22_C	4	192	757	580	14656	4118	22.01%	661	1266	16536	4587	35.91%	292	976	15882	4139	22.64%
C7552	2	104	108	105	1910	467	37.35%	96	117	1828	447	31.47%	78	111	1885	458	34.71%
des	2	128	245	129	3465	741	3.35%	245	185	3617	868	21.06%	131	146	3652	756	5.44%
g1296	7	186	1296	1123	35627	7999	54.30%	1296	4289	36873	9688	86.88%	1296	4289	37005	9676	86.65%
g216	2	108	216	109	4606	1017	56.94%	216	167	3483	820	26.54%	216	167	3487	820	26.54%
g625	4	157	625	475	12397	2975	19.10%	625	1607	15043	4330	73.34%	625	1607	14986	4316	72.78%
hyp	2	128	128	129	213748	45435	0.65%	128	256	145628	29805	-33.98%	128	256	145628	29805	-33.98%
i2	2	101	1	102	514	165	170.49%	1	10	161	47	-22.95%	1	10	159	47	-22.95%
i10	2	129	224	130	2133	692	36.49%	180	224	2365	740	45.96%	128	158	2195	662	30.57%
max	3	171	130	345	3932	1141	40.52%	130	395	3912	1003	23.52%	129	390	3894	993	22.29%
mem_ctrl	7	172	1231	1039	19982	6372	22.37%	772	3294	27465	8317	59.73%	388	2206	25150	7500	44.04%
voter	6	167	1	841	14772	2511	50.63%	1	166	11446	1921	15.24%	1	166	11389	1899	13.92%
avg.							**46.59%**					**34.84%**					**25.08%**

circuit	#frm	#in	structural-f					structural-sf				
			#out	#FF	#gate	#LUT	overhead	#output	#FF	#gate	#LUT	overhead
128-adder	2	128	65	2	624	210	-13.93%	65	2	624	210	-13.93%
b14_C	2	138	262	450	4299	1483	28.73%	189	347	4303	1391	20.75%
b15_C	3	162	274	432	7957	2315	17.75%	343	707	7927	2494	26.86%
b20_C	3	174	424	606	9723	2774	24.90%	248	364	9415	2491	12.16%
b21_C	3	174	424	598	9590	2813	21.72%	248	386	9497	2570	11.21%
b22_C	4	192	661	1100	15672	4425	31.11%	292	583	15078	3933	16.53%
C7552	2	104	96	117	1610	437	28.53%	78	111	1672	434	27.65%
des	2	128	245	185	3222	869	21.20%	131	146	3395	751	4.74%
g1296	7	186	1296	2420	35019	9130	76.12%	1296	2420	34936	9108	75.69%
g216	2	108	216	167	3180	808	24.69%	216	167	3155	794	22.53%
g625	4	157	625	1154	13033	3994	59.89%	625	1154	13045	4013	60.65%
hyp	2	128	128	256	146348	30328	-32.82%	128	256	146348	30328	-32.82%
i2	2	101	1	10	144	45	-26.23%	1	10	144	49	-19.67%
i10	2	129	180	224	1907	682	34.52%	128	158	1899	637	25.64%
max	3	171	130	218	3301	853	5.05%	129	219	3295	824	1.48%
mem_ctrl	7	172	772	2040	26287	7909	51.89%	388	1429	23881	7159	37.49%
voter	6	167	1	71	11199	1893	13.56%	1	71	11265	1880	12.78%
avg.							**29.14%**					**20.07%**

doi:10.6342/NTU202001058

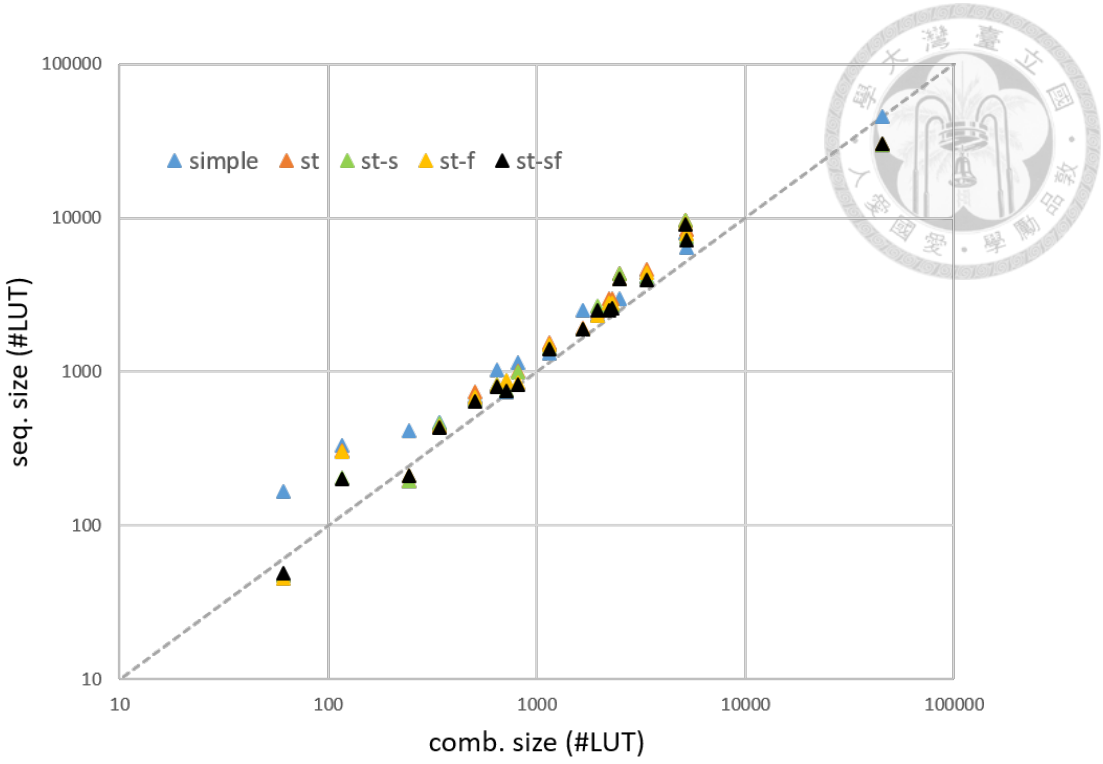
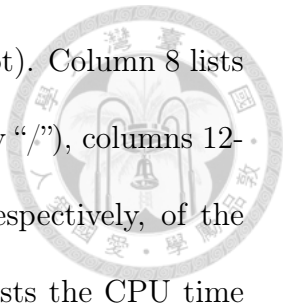


Figure 5.4: Circuit size after structural folding.

5.2.2 Comparing Structural and Functional Folding

To compare the performance of the structural and functional methods, we conducted experiments on 11 benchmarks, each being folded by 4, 8 and 16 time-frames. A timeout limit of 300 seconds was imposed on pin scheduling and FSM construction combined (steps 1 and 2 in Figure 4.3), and the same limit was imposed on `MeMin` for state minimization (step 3 in Figure 4.3). Table 5.6 shows the 33 results, where columns 2-4 list the folding number and the numbers of input/outputs of the folded sequential circuits, respectively, and columns 5-15 list the folded circuit information of the two methods, including the number of outputs, AIG nodes, LUTs, and flip-flops. The results of the functional method are annotated in column 14 with the applied configurations: whether to enable input reordering (r/nr), whether to mini-

doi:10.6342/NTU202001058



mize FSM states (m/nm), and the two encoding options (nat/1hot). Column 8 lists the numbers of states before and after minimization (separated by "/"), columns 12-13 list the reduction on the numbers of LUTs and flip-flops, respectively, of the functional method over the structural method, and column 15 lists the CPU time in seconds of the functional method on each benchmark. An entry "-" in the table indicates that the value cannot be obtained within the timeout limit. The structural method took less than a second for all the experiments, while the functional method generated results for 29 of the 33 instances within the timeout limit. On the other hand, the functional method achieved an average of 40.40% and 33.74% reductions on LUT and flip-flop usage, respectively, over the structural method in the 29 cases.

In addition, we compared the sizes of the original combinational circuits to their folded sequential circuits under the two methods in terms of the number of LUTs. The results are plotted in Figure 5.5, where the triangular and circular points correspond to the results of the structural and functional methods, respectively, and the blue, green, and orange points correspond to results folded by 16, 8 and 4 timeframes, respectively. The data points to the right of the gray dotted line are the cases where the folded circuits are smaller than their combinational counterparts. It is interesting to note that 20 of the 29 results obtained by the functional method achieved circuit size reduction, while 26 of the 33 results from the structural method incurred LUT overhead. The overhead of the structural method is understandable because circuit folding introduces additional control logic and flip-flop boundaries to the original circuit that restricts combinational synthesis.

From the experimental results in Table 5.6, we notice that the functional method

doi:10.6342/NTU202001058

Table 5.6: Comparison between structural and functional methods.

circuit	#frm	#in	#out	structural-sf			functional							
				#gate	#LUT	#FF	#state	#gate	#LUT	#FF	#LUT redu.	#FF redu.	config	runtime
64-adder	16	8	5	56	29	17	32/2	32	7	1	75.86%	94.12%	nr/m/nat	0.28
	8	16	9	98	35	9	16/-	150	40	4	-14.29%	55.56%	nr/nm/nat	9.29
	4	32	17	205	82	5	-	-	-	-	-	-	-	>300
128-parity	16	8	1	41	23	17	32/2	24	3	1	86.96%	94.12%	r/m/nat	0.16
	8	16	1	60	15	9	16/-	74	9	4	40.00%	55.56%	r/nm/nat	5.80
	4	32	1	104	15	5	-	-	-	-	-	-	-	>300
apex2	16	3	1	2409	836	338	474/-	1764	734	474	12.20%	-40.24%	r/nm/1hot	0.38
	8	5	2	2135	736	240	327/-	1767	696	327	5.43%	-36.25%	r/nm/1hot	0.13
	4	10	3	1703	696	236	127/-	1177	444	127	36.21%	46.19%	r/nm/1hot	0.12
arbiter*	16	16	1	877	274	144	47/4	53	12	2	95.62%	98.61%	r/m/nat	0.57
	8	32	1	836	244	136	23/4	104	25	2	89.75%	98.53%	r/m/nat	0.53
	4	64	1	800	232	132	11/4	165	47	2	79.74%	98.48%	r/m/nat	0.51
b17_C*	16	24	1	2263	636	142	233/-	1149	472	232	25.79%	-63.38%	r/nm/1hot	42.89
	8	48	1	2029	566	134	86/-	746	279	86	50.71%	35.82%	r/nm/1hot	85.98
	4	95	1	1651	465	114	-	-	-	-	-	-	-	>300
bcb	16	2	5	2573	852	379	533/-	1460	848	533	0.47%	-40.63%	nr/nm/1hot	0.06
	8	4	8	2014	750	324	273/227	1086	509	273	32.13%	15.74%	nr/nm/1hot	0.05
	4	7	20	1938	760	367	141/99	940	375	141	50.66%	61.58%	nr/nm/1hot	0.08
e64	16	5	5	389	142	37	29/14	74	17	4	88.03%	89.19%	r/m/nat	0.12
	8	9	9	349	122	28	16/9	108	26	4	78.69%	85.71%	r/nm/nat	0.08
	4	17	17	294	115	22	8/-	162	52	3	54.78%	86.36%	r/nm/nat	4.68
i2	16	13	1	409	132	43	54/-	207	87	6	34.09%	86.05%	r/nm/nat	0.25
	8	26	1	329	96	31	25/-	152	61	25	36.46%	19.35%	r/nm/1hot	0.18
	4	51	1	245	73	23	14/-	130	48	14	34.25%	39.13%	r/nm/1hot	0.22
i3	16	9	1	171	68	24	40/-	145	62	35	8.82%	-45.83%	r/nm/1hot	0.09
	8	17	1	116	43	16	22/-	103	41	20	4.65%	-25.00%	r/nm/1hot	0.12
	4	33	2	106	36	11	10/-	89	32	9	11.11%	18.18%	r/nm/1hot	29.05
i4	16	12	1	439	132	39	83/-	458	184	82	-39.39%	-110.26%	r/nm/1hot	3.85
	8	24	1	338	94	27	38/-	295	124	37	-31.91%	-37.04%	r/nm/1hot	5.48
	4	48	2	233	60	15	-	-	-	-	-	-	-	>300
too_large	16	3	1	4331	1475	615	305/-	1057	470	305	68.14%	50.41%	r/nm/1hot	0.14
	8	5	2	3975	1387	541	187/-	805	331	187	76.14%	65.43%	r/nm/1hot	0.11
	4	10	3	3245	1284	537	92/-	673	250	92	80.53%	82.87%	r/nm/1hot	0.11
avg.											**40.40%**	**33.74%**		

doi:10.6342/NTU202001058

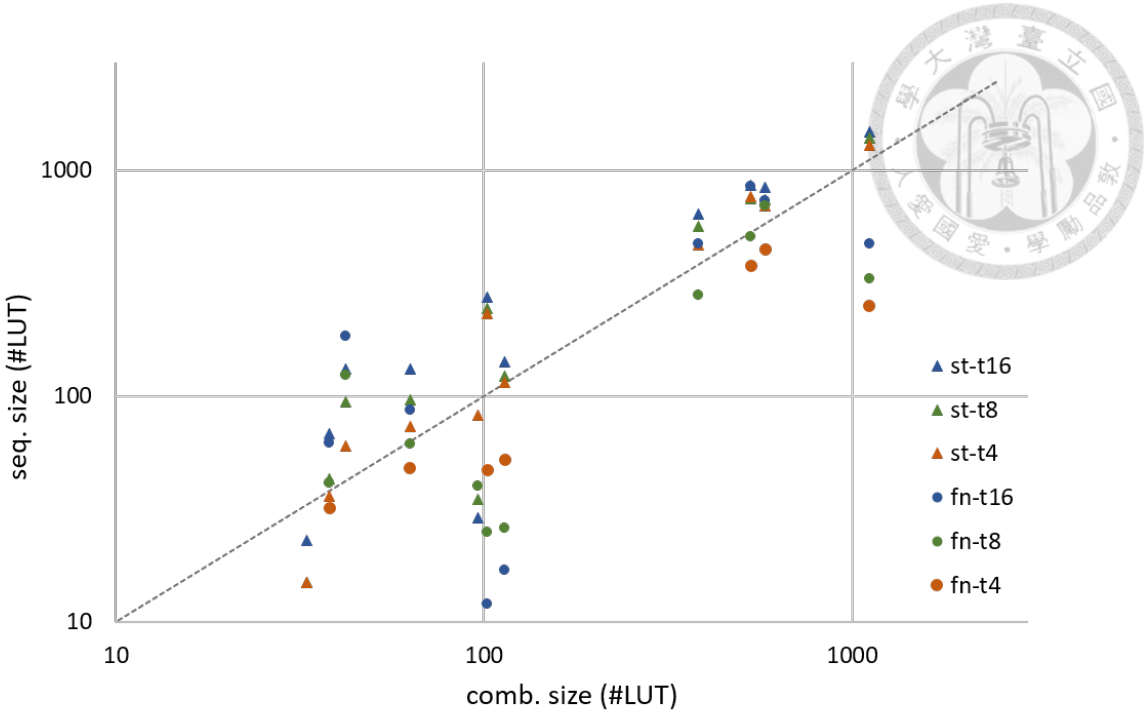
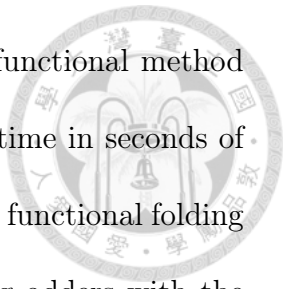


Figure 5.5: Circuit size comparison between structural and functional folding.

performs especially well on certain types of circuits with intrinsic iterative structures, such as `64-adder` and `128-parity`. Since the original TFF algorithm is designed for iterative circuits, the phenomenon is conceivable. We further conducted experiments on adders and majority voters. For an n-bit adder with $2n$ primary inputs and $n+1$ primary outputs, it is folded by n time-frames, resulting in a sequential circuit with 2 input pins and 2 output pins. Similarly, for an n-bit majority voter with n primary inputs and 1 primary output, it is folded by n time-frames, resulting in a sequential circuit with 1 input pin and 1 output pin. The results of folding adders and voters are shown in Table 5.7 and Table 5.8. Columns 2-3 of the 2 tables list the size information of the original combinational circuit, column 4 lists the folding number, and columns 5-15 list the folded circuit information of the two methods, including the number of AIG nodes, LUTs, and flip-flops. Column 8 lists the numbers of states before and after minimization (separated by "/"), and columns 12-13 list the reduc-

doi:10.6342/NTU202001058



tion on the numbers of LUTs and flip-flops, respectively, of the functional method over the structural method Finally, columns 14-15 list the CPU time in seconds of the FSM construction step and the FSM minimization step during functional folding on each benchmark, respectively. The FSMs obtained by folding adders with the functional method all reduced to the FSM of the 1-bit carry-save adder shown in Figure 4.4b. Therefore, the circuit size remains the same after functional folding. The results demonstrate the circuit size compaction ability of the functional method, as the numbers of LUTs of functionally folded circuits are significantly smaller than those of structurally folded circuits and original combinational circuits. However, the results also signify the limitation of the functional method in computation time. The FSM of the `256-adder` could not be constructed within 10 hours, since this step relies on BDD-based operations, which could be time-consuming for larger circuits. On the other, while the FSM construction could be done pretty fast for the voters, the FSM minimization of `23-voter` and `25-voter` took over 5 hours to compute, as `MeMin` struggled to find a minimum-state FSM. Despite the 2 computational bottlenecks in the FSM construction and minimization steps, the functional method can usually obtain a more optimal folded circuit when compared to the structural method in our experiments.

5.2.3 Case Study of Combining Structural and Functional Folding

From the experimental results, the structural method demonstrates its effectiveness and scalability in folding large circuits, while the functional method generates better

doi:10.6342/NTU202001058

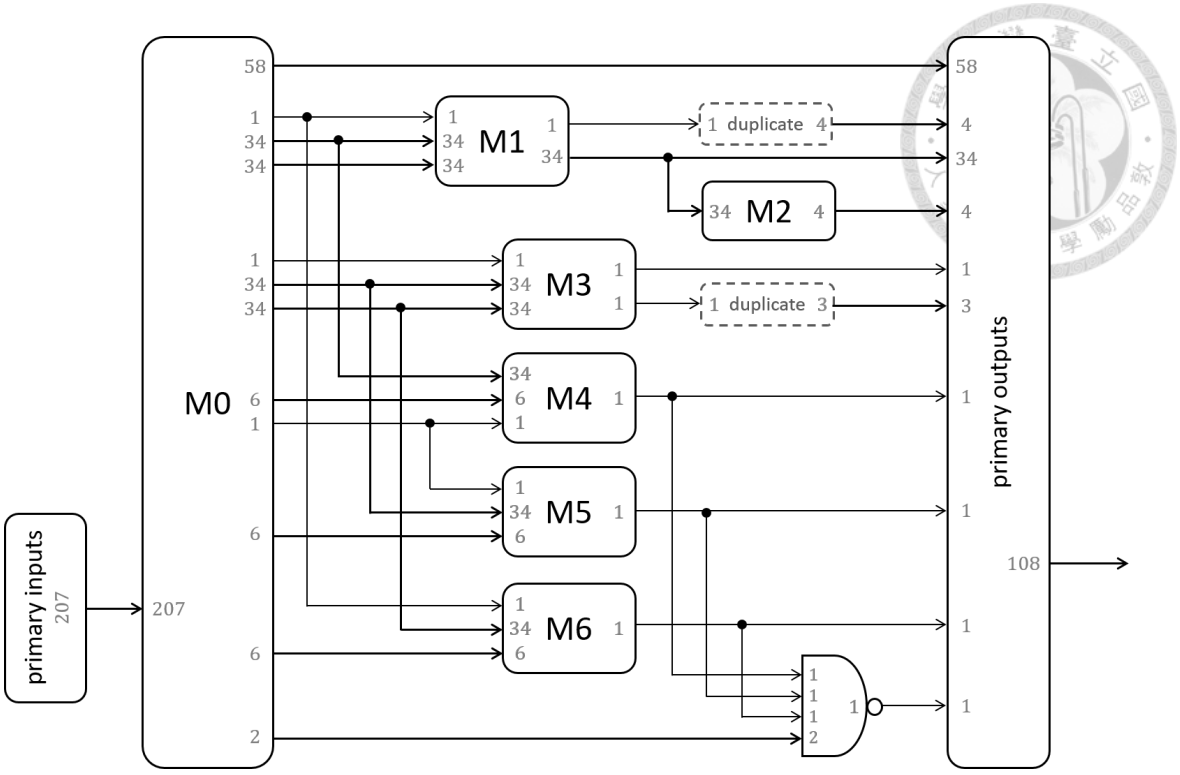
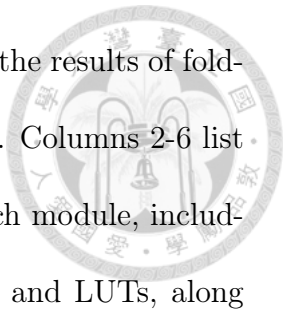


Figure 5.6: Hierarchical structure of C7552.

solutions with a higher computation cost. To show that the combination of the 2 methods can produce a more optimal result, we conducted a case study on circuit C7552, which cannot be directly folded with functional method under runtime limit. The high-level hierarchical structure of C7552 is shown in Figure 5.6, where each module or subcircuit is represented by a rounded rectangle, and the bit-width of each wire is annotated in gray on the edge of the connecting modules.

The design of each module was processed and synthesized into a combinational circuit by Yosys [37], and then be folded by 2 time-frames with either the structural or functional method. During the folding procedure, the input schedule of a module is constrained by the output schedule of its fanin modules, that is, the input signal of a module should be scheduled at a time-frame no earlier than the time-frame it is produced as an output by other modules. After the 7 modules of C7552 are folded,

doi:10.6342/NTU202001058



they are connected into an overall folded circuit. Table 5.9 shows the results of folding each module and the statistics of the combined folded circuit. Columns 2-6 list the information of the combinational circuit synthesized from each module, including the number of primary inputs, primary outputs, AIG nodes and LUTs, along with its functionality. Columns 7-13 list the information of the folded sequential circuit, including the number of input pins, output pins, AIG nodes, LUTs, and the LUT overhead incurred comparing to the original combinational circuit, with the last column listing the corresponding folding method. The last row of Table 5.9 lists the information of the `C7552` circuit, when being structurally folded with the same input/output schedule as the combined folded circuit. The results show that the combination of the 2 folding methods can indeed generate a better folded circuit when compared to the one folded only with the structural method, with 55.26% and 28.81% reduction in flip-flop and LUT usage, respectively. Therefore, by combing the structural and functional methods, we can achieve higher scalability with an improved optimality in the resulting folded circuits.

doi:10.6342/NTU202001058

Table 5.7: Results of folding adders.

circuit	original		#frm	structural-sf			functional							
	#gate	#LUT		#gate	#LUT	#FF	#state	#gate	#LUT	#FF	#LUT redu.	#FF redu.	runtime	
													FSM constr.	MeMin
8-adder	56	14	8	20	12	9	16/2	8	3	1	300.00%	800.00%	0.024	0.022
16-adder	112	31	16	28	26	17	32/2	8	3	1	766.67%	1600.00%	0.03	0.022
32-adder	231	61	32	44	44	33	64/2	8	3	1	1366.67%	3200.00%	0.051	0.022
64-adder	507	96	64	80	84	65	128/2	8	3	1	2700.00%	6400.00%	0.112	0.024
128-adder	844	244	128	140	168	129	256/2	8	3	1	5500.00%	12800.00%	0.289	0.024
256-adder	1894	487	256	268	336	257	-	-	-	-	-	-	>36000	-
avg.											**2126.67%**	**4960.00%**		

Table 5.8: Results of folding voters.

circuit	original		#frm	structural-sf			functional							
	#gate	#LUT		#gate	#LUT	#FF	#state	#gate	#LUT	#FF	#LUT redu.	#FF redu.	runtime	
													FSM constr.	MeMin
15-voter	102	14	15	255	83	32	79/9	17	5	4	93.98%	87.50%	0.03	0.30
17-voter	142	14	17	268	88	34	98/10	15	5	4	94.32%	88.24%	0.03	1.51
19-voter	175	22	19	329	102	38	119/11	15	5	4	95.10%	89.47%	0.03	63.56
21-voter	211	22	21	371	119	40	142/12	14	5	4	95.80%	90.00%	0.03	985.64
23-voter	199	22	23	420	129	42	167/13	16	5	4	96.12%	90.48%	0.03	21526.34
25-voter	254	22	25	473	141	47	194/14	15	5	4	96.45%	91.49%	0.03	412215.83
avg.											**95.29%**	**89.53%**		

Table 5.9: Results of folding `C7552` with the structural and functional methods combined.

circuit	original					folded						
	#PI	#PO	#gate	#LUT	description	#in	#out	#FF	#gate	#LUT	overhead	method
M0	207	217	330	217	bus signal controller	104	111	5	506	158	-27.19%	structural
M1	69	35	298	78	34-bit adder	35	18	2	128	33	-57.69%	functional
M2	34	4	168	12	sum parity checker	17	3	2	62	9	-25.00%	functional
M3	69	2	144	40	34-bit comparator	35	2	2	72	26	-35.00%	functional
M4	42	1	195	15	sanity checker	21	1	3	87	13	-13.33%	functional
M5	42	1	195	15	sanity checker	21	1	3	87	13	-13.33%	functional
M6	42	1	195	15	sanity checker	21	1	3	92	14	-6.67%	functional
C7552	207	108	1485	340	overall circuit	104	64	17	946	257	-24.41	combined
						104	64	38	1658	361	6.18	structural

doi:10.6342/NTU202001058

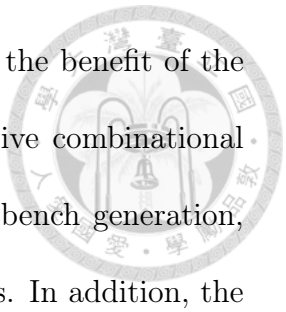
Chapter 6

Conclusions and Future Work

6.1 Conclusions

In the thesis, we have introduced circuit folding as a process of transforming a combinational circuit $\mathcal{C}_C$ into a sequential circuit $\mathcal{C}_S$, which after time-frame expansion, becomes functionally equivalent to $\mathcal{C}_C$. We have formulated the time-frame folding problem, and provided a computational solution based on functional decomposition for state identification and transition reconstruction. Our proposed algorithm guarantees the sequential circuit folded from an iterative combinational circuit is state minimized. We have further extended the concept of folding for general combinational circuits and formulated a circuit folding approach to time multiplexing on FPGAs. The structural and functional methods, orthogonal to prior time multiplexing methods, have been proposed and implemented to show their potentials to alleviate the I/O-pin bottleneck of FPGAs. Circuit folding can be applied to var-

doi:10.6342/NTU202001058

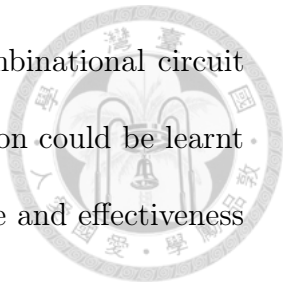


ious tasks in logic synthesis. Experimental results demonstrated the benefit of the time-frame folding method in circuit compaction from an iterative combinational circuit to its sequential counterpart, which can be useful in testbench generation, sequential synthesis of bounded strategies, and other applications. In addition, the experiments on time multiplexing suggested the scalability of the structural method and the optimization power of the functional method. From the case study of combining the 2 folding methods, we saw the potential of the hybrid method that can achieve both scalability and optimality.

6.2 Future Work

For future work, since the finite state machine (FSM) shares a lot of similarities with the finite state automata (FSA), we would like to extend the time-frame folding algorithm for applications in automata theory. Given a set of symbolic constraints of bounded-length strings describing a regular language L, i.e. the characteristic functions of the accepting (or rejecting) strings, the time-frame folding algorithm, with slight modification, should be able to derive a symbolic finite automaton [34] complying with the language L, with the transition condition (depicted in Subsection 3.2.2) serving as the predicate of the transition between 2 states. We would like to investigate the applicability of the above-described method in the context of finite automata learning, such as in [2,13], a finite (symbolic) automaton is learnt by membership queries and conjectures from an oracle, or as in [10], a separating finite automaton of 2 languages is learnt with a similar manner. On the other hand, we would also like to apply some automata learning procedure, e.g. the $L^\star$ algorithm [2],

doi:10.6342/NTU202001058



to the problem of time-frame folding. The given k-iterative combinational circuit would serve as the oracle or the teacher, from which an automaton could be learnt correspondingly. We would then like to compare the performance and effectiveness of such method with our proposed BDD-based algorithm.

For another future work, we would like to fully automate the hybrid folding method of combining the structural and functional method for time multiplexing, especially in the circuit partitioning stage. In the case study we conducted, we relied on the given high-level hierarchical design and partitioned the circuit into smaller modules manually. Therefore, it would be more desirable if the partitioning could be done automatically from a flattened gate-level logic netlist. Moreover, we would like to investigate other functional decomposition techniques to help mitigate the high computational cost of BDD-based operations during time-frame and functional circuit folding.

doi:10.6342/NTU202001058

Bibliography

[1] A. Abel and J. Reineke. MEMIN: SAT-based exact minimization of incompletely specified Mealy machines. In *Proceedings of International Conference of Computer-Aided Design (ICCAD)*, pages 94–101, 2015.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 11 1987.

[3] R. Ashenhurst. *The Decomposition of Switching Functions*, volume 29, pages 74–116. Computation Lab, Harvard University, 1959.

[4] J. Babb, R. Tessier, and A. Agarwal. Virtual wires: Overcoming pin limitations in FPGA-based logic emulators. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, 1993.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207, 1999.

[6] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

doi:10.6342/NTU202001058

[7] R. Brayton and A. Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 24–40, 2010.

[8] S.-C. Chang, M. Marek-Sadowdka, and T. Hwang. Technology mapping for TLU FPGAs based on decomposition of binary decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 15(10):1226–1236, 1996.

[9] S.-C. Chen, R. Sun, and Y.-W. Chang. Simultaneous partitioning and signals grouping for time-division multiplexing in 2.5D FPGA-based systems. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 2018.

[10] Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFA's for compositional verification. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 31–45, 2009.

[11] P.-C. Chien and J.-H. Jiang. Time-frame folding: Back to the sequentiality. In *Proceedings of International Conference of Computer-Aided Design (ICCAD)*, 2019.

[12] P.-C. Chien and J.-H. Jiang. Time multiplexing via circuit folding. In *Proceedings of Design Automation Conference (DAC)*, 2020.

[13] S. Drews and L. D'Antoni. Learning symbolic automata. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 173–189, 2017.

doi:10.6342/NTU202001058

[14] I. Han and Y. Shin. Folded circuit synthesis: Min-area logic synthesis using dual-edge-triggered flip-flops. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23:1–21, 08 2018.

[15] S. Hauck and G. Borriello. Pin assignment for multi-FPGA systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 16(9):956–964, 1997.

[16] W. N. Hung and R. Sun. Challenges in large FPGA-based logic emulation systems. In *Proceedings of International Symposium on Physical Design (ISPD)*, pages 26–33, 2018.

[17] J.-H. R. Jiang and R. K. Brayton. On the verification of sequential equivalence. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 22(6):686–697, 2003.

[18] J.-H. R. Jiang, J.-Y. Jou, and J.-D. Huang. Compatible class encoding in hyper-function decomposition for FPGA synthesis. In *Proceedings of Design Automation Conference (DAC)*, pages 712–717, 1998.

[19] N. Kushik, J. López, A. Cavalli, and N. Yevtushenko. Improving protocol passive testing through "gedanken" experiments with finite state machines. In *Proceedings of International Conference on Software Quality, Reliability and Security (QRS)*, pages 315–322, 2016.

[20] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula. Bdd based decomposition of logic functions with application to FPGA synthesis. In *Proceedings of Design Automation Conference (DAC)*, pages 642–647, 1993.

doi:10.6342/NTU202001058

[21] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[22] H. Liu and D. F. Wong. Network flow based circuit partitioning for time-multiplexed FPGAs. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 497–504, 1998.

[23] S. Liu, F. Lau, and B. Carrion Schafer. Investigation and optimization of pin multiplexing in high-level synthesis. In *Proceedings of Great Lakes Symposium on VLSI (GLSVLSI)*, pages 427–430, 2018.

[24] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla. GLA: Gate-level abstraction revisited. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1399–1404, 2013.

[25] A. Myaing and V. Dinavahi. FPGA-based real-time emulation of power electronic systems with detailed representation of device characteristics. *IEEE Transactions on Industrial Electronics (TIE)*, 58:358 – 368, 2011.

[26] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of International Conference of Computer-Aided Design (ICCAD)*, pages 628–631, 1994.

[27] K. Parhi, C.-Y. Wang, and A. Brown. Synthesis of control circuits in folded pipelined DSP architectures. *IEEE Journal of Solid-State Circuits (JSSC)*, 27:29 – 43, 02 1992.

doi:10.6342/NTU202001058

[28] J.-K. Rho, F. Somenzi, and C. Pixley. Minimum length synchronizing sequences of finite state machine. In *Proceedings of Design Automation Conference (DAC)*, pages 463–468, 1993.

[29] J. P. Roth and R. M. Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, 6(2):227–238, 1962.

[30] S. Sandberg. Homing and synchronizing sequences. In *Model-Based Testing of Reactive Systems: Advanced Lectures*, pages 5–33. 2005.

[31] F. Somenzi. CUDD: CU decision diagram package (release 2.4.1). *University of Colorado at Boulder*, 2005.

[32] Q. Tang and M. Tuna. Multi-FPGA prototyping board issue: the FPGA I/O bottleneck. In *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2014.

[33] S. Trimberger. Scheduling designs into a time-multiplexed FPGA. In *Proceedings of International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 153–160, 1998.

[34] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of International Conference on Software Testing, Verification and Validation (ICST)*, pages 498–507, 2010.

[35] H.-E. Wang, K.-H. Tu, J.-H. R. Jiang, and N. Kushik. Homing sequence derivation with quantified Boolean satisfiability. In *Proceedings of International Conference on Testing Software and Systems (ICTSS)*, pages 230–242, 2017.

doi:10.6342/NTU202001058

[36] L.-T. Wang, C.-W. Wu, and X. Wen. *VLSI Test Principles and Architectures: Design for Testability (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., 2006.

[37] C. Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.

doi:10.6342/NTU202001058