NAIST-IS-DT9861205

Doctor's Thesis

Studies on Multi-Cycle Paths of Sequential Circuits

Kazuhiro Nakamura

February 5, 2001

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

Doctor's Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
DOCTOR of ENGINEERING

Kazuhiro Nakamura

Thesis committee: Katsumasa Watanabe, Professor
Hideo Fujiwara, Professor
Akira Fukuda, Professor
Shinji Kimura, Associate Professor

Studies on Multi-Cycle Paths of Sequential Circuits[*]

Kazuhiro Nakamura

Abstract

The timing analysis and optimization of sequential circuits become more and more important with the advances of VLSI technologies. We should cope with the clock frequency more than 2 GHz near the future. The clock frequencies of sequential circuits are decided based on the maximum delay time of paths between clocked flip-flops under the assumption that all paths propagate signals within one clock cycle. The assumption is violated in many circuits including multi-cycle paths on which signals can be propagated with 2 or more clock cycles. At present, such multi-cycle paths are detected and manipulated by hands, but should be detected and manipulated automatically for the correct timing analysis.

In the thesis, we formalize multi-cycle paths and propose two detection methods of multi-cycle paths. We also describe the decision of the clock frequency, timing verification and logic optimization with considering the number of allowable clock cycles for signal propagations.

First, we focus on multi-cycle paths between flip-flops which are guarded with wait states, and formalize such multi-cycle paths based on update cycle of flip-flops.

Second, we propose a detection method of multi-cycle paths based on a symbolic state traversal of FSMs (Finite State Machines) using BDD's (Binary Decision Diagrams). The method detects multi-cycle paths based on the update cycle analysis of flip-flops. The method computes reachable states from the initial state of FSM using symbolic state traversal of FSM. Then the method computes allowable clock cycles of each path between flip-flops and detects multi-cycle paths.

[*]Doctor's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9861205, February 5, 2001.

We have applied our method to 30 ISCAS89 benchmarks and found that 22 circuits include multi-cycle paths. 11 circuits among them include multi-cycle paths which are critical paths, where the delay is measured as the number of gates on the path. Experimental results show that many multi-cycle paths exist in sequential circuits.

Third, we propose a detection method of multi-cycle paths based on propositional satisfiability (SAT). The method generates a CNF (Conjunctive Normal Form) formula which is satisfiable if and only if the path is not a multi-cycle path, and checks the satisfiability of the formula with a SAT prover. In the conversion from multi-level circuits into CNF formulae, the size of resulting CNF formulae may grow exponentially. To reduce the size of CNF formulae, we introduce heuristics on the conversion. The set of reachable states is hard to manipulate based on SAT based-methods, so we adopt an approximation that all states are reachable from the initial state. By this approximation, some multi-cycle paths may not be detected, but there are many detectable multi-cycle paths in large circuits. We have applied our method to ISCAS89 benchmarks and other sample circuits. Experimental results show the improvement in the size of manipulatable circuits, especially hard circuits for BDD manipulation, and show 54 % of multi-cycle paths can be detected using the SAT-based method under the assumption that all states are reachable.

Fourth, we describe correct timing verification and logic optimization with the information of detected multi-cycle paths. In timing verification, we introduce the number of allowable clock cycles for each path, and check whether the clock period multiplied by the allowable cycles is greater than the delay time of the path. We applied the method to several circuits to decide the proper clock frequency. In logic optimization, the area of circuits is reduced by changing multi-cycle path parts to small circuits with long path delay, and by replacing gates on the path with small and slow gates.

Keywords:

Multi-Cycle Paths, Timing Verification, Sequential Circuits, Maximum Delay Analysis, Propositional Satisfiability, Formal Verification

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advances of VLSI technologies, the timing analysis and optimization of sequential circuits become more and more important. The clock frequencies of sequential circuits will become more than 2 GHz near the future. To cope with timing issue in designing high-speed sequential circuits, the advance of timing analysis system is demanded.

The clock frequencies of sequential circuits are decided based on the maximum delay time of paths between clocked flip-flops in the circuit. Hence, the precise estimation of the maximum delay time is important in deciding the proper clock frequency. The maximum delay can be computed as the longest path of the weighted graph corresponding to the circuit, where nodes in the graph are logic gates in the circuit and the weight of the node is the delay time of each gate [14].

In some case, such topological maximum delay paths cannot be sensitized with any input patterns and therefore become false paths. Many works have been done to detect false paths and to maximize the clock frequency [2, 6, 1].

On the other hand, in circuits, there exist paths which are sensitizable but do not affect the clock frequency. These paths are *multi-cycle paths* between clocked flip-flops. Multi-cycle paths are the paths whose input is stable for several clock cycles before the output is required, and 2 or more clock cycles are allowed for signal propagations. Hence, the delay time of multi-cycle paths can be greater than the clock period.

The clock frequencies of sequential circuits are generally decided under the assumption that all paths propagate signals within one clock cycle. However, the assumption is violated in many circuits including multi-cycle paths. We should compute the number of allowable clock cycles for each path, and check whether the clock period multiplied by the allowable cycles is greater than the delay time of the path. At present, multi-cycle paths are detected and manipulated by hands, but should be detected and manipulated automatically for the correct timing analysis.

A timing verification technique which can handle multi-cycle paths in microprocessors has been developed [10]. The method generates state transition graph (STG) of the controller of microprocessor and checks the specified timing constraints by manipulating the relation of events represented by inequalities on delays of modules. However, the method aims at microprocessor-based designs, and is hard to apply to gate level circuits because STGs of gate level circuits are too large to be represented.

In the thesis, we handle gate level circuits using finite state machine (FSM) model of sequential circuits, and automatically detect multi-cycle paths. We discuss the properties of multi-cycle paths and propose two detection methods of multi-cycle paths. First, we propose a detection method of multi-cycle paths based on a symbolic state traversal of FSMs using binary decision diagrams (BDD's) [7, 5]. Then, we propose a propositional satisfiability (SAT) based detection method of multi-cycle paths. SAT is a method to decide whether a Boolean formula is satisfiable or not, and recently applied to the analysis of sequential circuits, such as symbolic model checking [3] and timing analysis of combinational circuits [20]. We also describe the decision of the clock frequency, timing verification and logic optimization with considering the number of allowable clock cycles for signal propagations.

In Chapter 2, we focus on multi-cycle paths between clocked flip-flops which are guarded with wait states. We define multi-cycle paths and propose a detection method of multi-cycle paths based on a symbolic state traversal of FSMs using BDD's. The method detects multi-cycle paths based on the update cycle analysis of flip-flops. The method computes reachable states from the initial state of FSM using BDD's. Then the method computes allowable clock cycles of each path between flip-flops and detects multi-cycle paths. We have applied our method to 30 ISCAS89 benchmarks and found that 22 circuits include multi-cycle paths.

11 circuits among them include multi-cycle paths which are critical paths, where the delay is measured as the number of gates on the path. Experimental results show that many multi-cycle paths exist in sequential circuits.

In Chapter 3, we propose a SAT based detection method of multi-cycle paths. The method generates a conjunctive normal form (CNF) formula which is satisfiable if and only if the path is not a multi-cycle path, and checks the satisfiability of the formula with a SAT prover. In the conversion from multi-level circuits into CNF formulae, the size of resulting CNF formulae may grow exponentially. To reduce the size of CNF formulae, we introduce heuristics on the conversion. The set of reachable states is hard to manipulate with SAT based-methods, so we adopt an approximation that all states are reachable from the initial state. By this approximation, some multi-cycle paths may not be detected, but there are many detectable multi-cycle paths in large circuits. We have applied our method to ISCAS89 benchmarks and other sample circuits. Experimental results show the improvement in the size of manipulatable circuits, especially hard circuits for BDD manipulation, and show 54 % of multi-cycle paths can be detected using the SAT-based method under the assumption that all states are reachable.

In Chapter 4, we describe logic optimization with the information of detected multi-cycle paths. In logic optimization, the area of circuits is reduced by changing multi-cycle path parts to small circuits with long path delay, and by replacing gates on the path with small and slow gates.

Chapter 2

Multi-Cycle Path Analysis Based on Symbolic State Traversal of FSM

2.1. Introduction

The clock frequency of a sequential logic circuit is decided based on the maximum delay of the combinational parts of the circuit. The precise estimation of the maximum delay is important in deciding the proper clock frequency. The maximum delay can be computed as the longest path of the weighted graph corresponding to the circuit, where nodes in the graph are logic gates in the circuit and the weight of the node is the delay time of each gate. The computational complexity of the maximum delay computation is proportional to the size of the graph.

In some case, such topological maximum delay paths cannot be sensitized with any input patterns and therefore become false paths. Many researches have been done to detect false paths and to minimize the clock period for obtaining the maximum clock frequency [2, 6, 1]. False paths are classified into combinational false paths and sequential false paths. Combinational false paths are the paths where we have no input patterns to sensitize these paths. Sequential false paths are the paths which can be sensitized only by unreachable states of the circuit. Note that these combinational and sequential false paths are non-sensitizable paths.

In logic circuits, there may exist paths which are sensitizable but do not affect the clock period. Within such paths, we focus on multi-cycle paths where 2 or more clock cycles are allowed for signal propagations. We describes properties of multi-cycle paths and propose a detection method based on Finite State Machine (FSM) model of logic circuits.

The rest of this chapter is organized as follows. In Section 2.2, we show preliminaries. Section 2.3 shows an example of a multi-cycle path and Section 2.4 describes the update cycle analysis of flip-flops. In Section 2.5, we show the detection method of multi-cycle paths. Experimental results are shown in Section 2.6.

2.2. Finite State Machines

Here, we show a definition of a finite state machine (FSM) based on [11] for the analysis of multi-cycle paths.

Definition 2.1 An FSM M is a 6-tuple $(S,\ \Sigma,\ \Gamma,\ \delta,\ \lambda,\ q_0)$ where

- S is a finite set of states,
- Σ is an input alphabet,
- Γ is an output alphabet,
- $\delta : S \times \Sigma \rightarrow S$ is a state transition function,
- $\lambda : S \times \Sigma \rightarrow \Gamma$ is an output function,
- $q_0 (\in S)$ is the initial state.

□

The behavior of $M = (S,\ \Sigma,\ \Gamma,\ \delta,\ \lambda,\ q_0)$ with respect to an input sequence $a_1 a_2 \ldots \ldots a_n$ $(a_i \in \Sigma)$ is a sequence of states $q_0 q_1 \ldots q_n$ $(q_i \in S)$ and a sequence of outputs $o_1 o_2 \ldots o_n$ $(o_i \in \Gamma)$ satisfying $q_i = \delta(q_{i-1},\ a_i)$ and $o_i = \lambda(q_{i-1},\ a_i)$.

Let Σ^* be a set of all input sequences over Σ, and let Σ^k be a set of input sequences with length k. We use ε as the sequence of length 0.

To represent the behavior of M, the domain of δ and λ are extended, and $\delta^* :\ S \times \Sigma^* \rightarrow S$ and $\lambda^* :\ S \times \Sigma^* \rightarrow \Gamma^*$ are introduced.

Definition 2.2 $\delta^* : S \times \Sigma^* \rightarrow S$ is defined as follows:

- $\delta^*(q, \varepsilon) = q$
- $\delta^*(q, xa) = \delta(\delta^*(q, x), a), \ (a \in \Sigma, x \in \Sigma^*)$

$\square$

Definition 2.3 $\lambda^* : S \times \Sigma^* \rightarrow \Gamma^*$ is defined as follows:

- $\lambda^*(q, \varepsilon) = \varepsilon$
- $\lambda^*(q, xa) = \lambda^*(q, x) \cdot \lambda(\delta^*(q, x), a), \ (a \in \Sigma, x \in \Sigma^*)$

$\square$

Note that real sequential logic circuits consist of flip-flops and combinational parts, and the flip-flops are modeled as a state in the FSM model. In other words, a state of the FSM model corresponds to the tuple of these values of all flip-flops at each time, and λ corresponds to a tuple of values of primary outputs of sequential circuits. In the following, we focus on the values of specific flip-flops, and use λ_r to denote the output function representing the value of a flip-flop r. It should be also noted that we deal only with edge-triggered flip-flops, but not with transparent latches.

[Example]

Consider a 4-bit synchronous up counter shown in Figure 2.1. The counter consists of 4 flip-flops ff0, ff1, ff2 and ff3. The value of these flip-flops changes as (ff0, ff1, ff2, ff3) = (0, 0, 0, 0), (1, 0, 0, 0), (0, 1, 0, 0), ... synchronized with the clock signal when *in* is 1.

FSM of the counter is shown in Figure 2.2, where $S = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}\}$, and the initial state is q_0. Labels of transitions represent the value of the input of the circuit *in* and the tuple of values of all flip-flops. For instance, 1/1000 represents that the value of the input is 1 and the tuple of values of flip-flops is (ff0, ff1, ff2, ff3) = (1, 0, 0, 0).

In the following, the 4-bit counter shown in Figure 2.1 and its FSM shown in Figure 2.2 are used to illustrate our multi-cycle paths detection method.

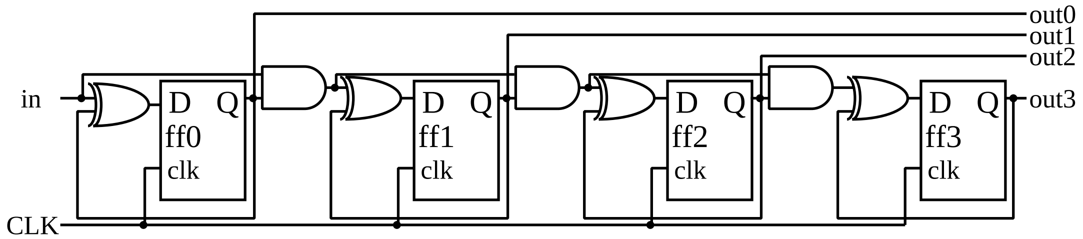


Figure 2.1. 4-bit synchronous counter.

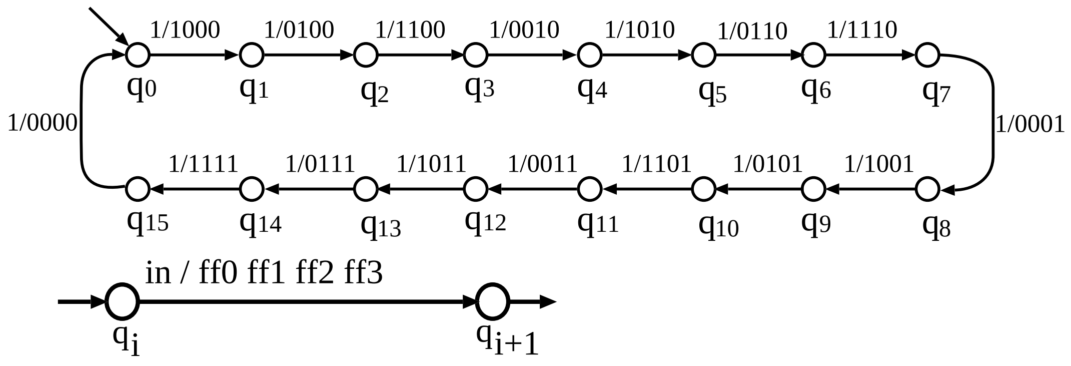


Figure 2.2. FSM of a 4-bit synchronous counter.

2.3. Multi-Cycle Paths in Sequential Circuits

Multi-cycle paths are the paths in combinational parts which are sensitizable but do not affect the decision of the clock period. Figure 2.3 shows an example of multi-cycle paths. The upper part of Figure 2.3 shows the data path, and the lower part is the control flip-flops. The initial state of the control flip-flops after the reset signal is (ff_q_3, ff_q_4, ff_q_5) = (1, 0, 0), and these flip-flops changes as (0, 1, 0), (0, 0, 1), (1, 0, 0), (0, 1, 0), ... synchronized with the clock signal.

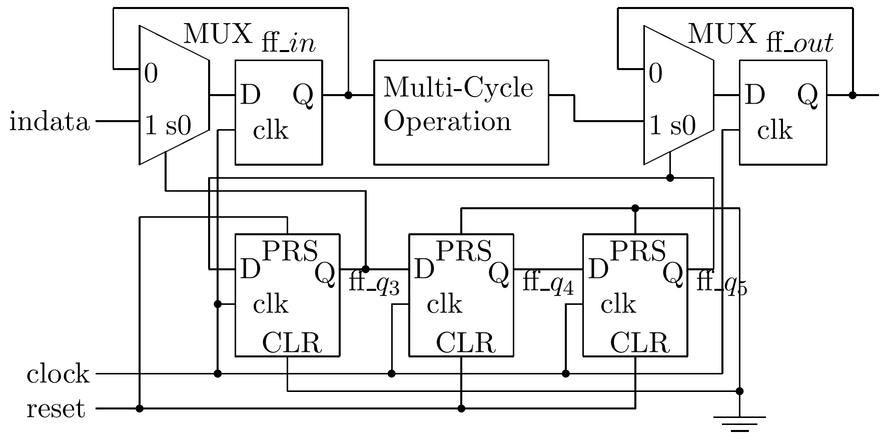


Figure 2.3. Example of a multi-cycle path.

At the data path, "ff_*in*" is set to "indata" when ff_q_3 = 1, and "ff_*out*" is set to the value of the output of "Multi-Cycle Operation" when ff_q_5 = 1. Since there is a wait state $(0, 1, 0)$, ff_q_5 is set to 1 two clocks after ff_q_3 is set to 1. Thus, there are 2 clock cycles for the computation of "Multi-Cycle Operation", and therefore the timing constraint of the path from "ff_*in*" to "ff_*out*" is

$$\text{(the path delay)} \;\leq\; 2 \times \text{(clock period)}$$

In general, the timing constraint of a path is

$$\text{(delay of the path)} \;\leq\; \{\text{(\#clocks)} \times \text{(clock period)}\}$$

where #clocks denotes the clock cycles usable to propagate signals along the path. #clocks of each path are computed as follows:

1. Update cycle analysis of flip-flops:
   we check whether the value of flip-flops has been changed or not at each state, and compute the update cycle.

2. Timing analysis of each path between flip-flops:
   we analyze the maximum allowable clock cycle of each path between flip-flops using the update cycle of the input and the output flip-flops.

In the following sections, we discuss update cycle analysis of flip-flops and timing analysis of each path between flip-flops.

Note that the set of reachable states from the initial state of FSM plays a key part on the multi-cycle path detection. For example, the reachable state set of the above example is $\{(1,0,0), (0,1,0), (0,0,1)\}$, and if all states can be reached, then the signal between ff_*in* and ff_*out* should be propagated within 1 cycle when $(1,1,1) \rightarrow (1,1,1)$.

2.4. Update Cycle Analysis of Flip-Flops

In the following, we focus on a flip-flop, and introduce a set of states where the value of the flip-flop does not change during k clocks.

First, let RS be the set of reachable states from the initial state of FSM.

Definition 2.4 (Reachable states from the initial state)

$$RS = \Big\{q \mid \exists x \in \Sigma^*,\ q = \delta^*(q_0,\, x)\Big\}$$

□

Let S_0 be the set of initial states of FSM. From our FSM definition $S_0 = \{q_0\}$. RS is obtained with the procedure shown in Figure 2.4. In Figure 2.4, PS represents the set of present states, and NS represents the set of next states of PS. States are traversed until newly visited states becomes $\emptyset$.

Note that the procedure is executed using a symbolic state traversal of finite state machines using BDD's (Binary Decision Diagrams) [9, 7, 5]. In the symbolic state traversal, primary inputs and flip-flops are represented as logical variables, and S_0, PS, NS, RS, δ are all described as logic functions or BDD's representing

```
ReachableStates($S_0$)
  $S_0$:  the set of the initial states of FSM
  $PS$:  the set of present states
  $NS$:  the set of next states of $PS$
  $RS$:  the set of reachable states from $S_0$
begin
  $PS$ = $S_0$;
  $RS$ = $S_0$;
  while ($PS \neq \emptyset$) do
    $NS$ = $\{\delta(q, a) \mid q \in PS,\ a \in \Sigma\}$;
    $PS$ = $NS - RS$;
    $RS$ = $RS \cup NS$;
  end while;
  return $RS$;
end
```

Figure 2.4. Analysis of reachable states from the initial state.

these functions. The manipulations of state sets such as $\cap$, $\cup$, $-$ are executed as logic operations such as AND, OR, and NOT.

Next, we define state sets RS^r_k ($\subseteq RS$) where the value of flip-flop r does not change during k clock cycles. In other words, when starting from a state in RS^r_k, the value of the flip-flop does not change for all input sequence with length k. Formally, state sets RS^r_k is defined as follows:

Definition 2.5 (State sets where flip-flop r does not change during k clock cycles)

$$RS^r_k \stackrel{\triangle}{=} \left\{q \mid q \in RS,\ \forall x \in \Sigma^k,\ \lambda^*_r(q, x) \in \{0^k,\ 1^k\}\right\}$$

□

The following property is satisfied on RS^r_k.

Property 2.1 (Property on RS^r_k)

$RS = RS^r_0 = RS^r_1 \supseteq RS^r_2 \supseteq RS^r_3 \supseteq RS^r_4 \ldots$

□

Note that, if there is a strongly connected component where the value of r is the same in any state, then $RS^r_k = RS^r_{k+1}$ with some k. Let $\mathcal{K}_r$ be the maximum number of k such that $RS^r_{k-1} \neq RS^r_k$.

Lemma 2.1 The following formula holds with $k > 2$.

$$RS^r_k = \{q \mid q \in RS^r_{k-1} \wedge \forall a \in \Sigma : \delta(q,a) \in RS^r_{k-1}\}$$

Proof: ($\supseteq$) Let q be an element of the set of the right side. Since $q \in RS^r_{k-1}$, $\lambda^*_r(q,x) \in \{0^{k-1},\ 1^{k-1}\}$ for all $x \in \Sigma^{k-1}$. Since $q' = \delta(q,a) \in RS^r_{k-1}$, $\lambda^*_r(q',x) \in \{0^{k-1},\ 1^{k-1}\}$ for all $x \in \Sigma^{k-1}$. Thus, for any $w \in \Sigma^k$, there exist $a \in \Sigma$ and $x \in \Sigma^{k-1}$ s.t. $w = ax$ and $\lambda^*_r(q,w) \in \{0^k,\ 1^k\}$. They say that $q \in RS^r_k$.
($\subseteq$) Let q be an element of the set RS^r_k, then $q \in RS^r_{k-1}$. From the definition of RS^r_k, for any $ax \in \Sigma^k$, $\lambda^*_r(\delta(q,a),x) \in \{0^{k-1},\ 1^{k-1}\}$. Hence, for any $a \in \Sigma$, $\delta(q,a) \in RS^r_{k-1}$. □

We show a procedure to compute the state sets RS^r_k based on Lemma 2.1.

1. Compute the set RS of reachable states from the initial state of FSM, and then let RS^r_1 be RS.

2. Using the state set RS, compute the state set RS^r_2, where $RS^r_2 = \{q \mid q \in RS,\ \forall a_1 a_2 \in \Sigma^2, \lambda_r(q,a_1) = \lambda_r(\delta(q,a_1),a_2)\}$.

3. Using RS^r_2, $RS^r_k (k \geq 3)$ is computed as follows. We also compute $\mathcal{K}_r$.

```
k = 3;
while($RS^r_{k-2} \neq RS^r_{k-1}$) do
  $RS^r_k = \{q | q \in RS^r_{k-1} \wedge \forall a \in \Sigma : \delta(q,a) \in RS^r_{k-1}\}$
  $k = k + 1$;
end while
$\mathcal{K}_r = k - 2$;
```

First, state sets RS^r_1 and RS^r_2 are computed using symbolic state traversal in step 1 and step 2. Second, state set RS^r_3 is computed from RS^r_2, and similarly, we can obtain state sets RS^r_4, RS^r_5, ... with Lemma 2.1 in step 3. Note that RS^r_k is computed until $RS^r_{k-2} = RS^r_{k-1}$, in some case $RS^r_{k-2} = RS^r_{k-1} = \emptyset$.

On the other hand, we define the set CS_r of states where the value of flip-flop r has just changed. Formally, state set CS_r is defined as follows:

Definition 2.6 (State set where flip-flop r has just changed)

$$CS_r = \{q \mid \exists a_1 a_2 \in \Sigma^2,\ q' \in RS,\ q = \delta(q', a_1),\ \lambda_r(q', a_1) \neq \lambda_r(q, a_2)\}$$

□

The CS_r can be computed similarly based on the symbolic state traversal.

[Example]

To illustrate update cycle analysis, consider a FSM shown in Figure 2.2. From the FSM, the set RS of reachable states of the FSM is $RS = \{q_0,\ q_1,\ q_2,\ q_3,\ q_4,\ q_5,\ q_6,\ q_7,\ q_8,\ q_9,\ q_{10},\ q_{11},\ q_{12},\ q_{13},\ q_{14},\ q_{15}\}$, and sets RS_1^{ff0}, RS_1^{ff1}, RS_1^{ff2} and RS_1^{ff3} are obtained as RS.

Next, let us consider sets RS_k^r of states where flip-flop r does not change during k clock cycles. Sets RS_k^r of the FSM are as follows.

$RS_2^{ff0} = \{\emptyset\}$.
$RS_2^{ff1} = \{q_1,\ q_3,\ q_5,\ q_7,\ q_9,\ q_{11},\ q_{13},\ q_{15}\}$
$RS_3^{ff1} = \{\emptyset\}$
$RS_2^{ff2} = \{q_0,\ q_1,\ q_3,\ q_4,\ q_5,\ q_7,\ q_8,\ q_9,\ q_{11},\ q_{12},\ q_{13},\ q_{15}\}$
$RS_3^{ff2} = \{q_0,\ q_3,\ q_4,\ q_7,\ q_8,\ q_{11},\ q_{12},\ q_{15}\}$
$RS_4^{ff2} = \{q_3,\ q_7,\ q_{11},\ q_{15}\}$
$RS_5^{ff2} = \{\emptyset\}$
$RS_2^{ff3} = \{q_0,\ q_1,\ q_2,\ q_3,\ q_4,\ q_5,\ q_7,\ q_8,\ q_9,\ q_{10},\ q_{11},\ q_{12},\ q_{13},\ q_{15}\}$
$RS_3^{ff3} = \{q_0,\ q_1,\ q_2,\ q_3,\ q_4,\ q_7,\ q_8,\ q_9,\ q_{10},\ q_{11},\ q_{12},\ q_{15}\}$
$RS_4^{ff3} = \{q_0,\ q_1,\ q_2,\ q_3,\ q_7,\ q_8,\ q_9,\ q_{10},\ q_{11},\ q_{15}\}$
$RS_5^{ff3} = \{q_0,\ q_1,\ q_2,\ q_7,\ q_8,\ q_9,\ q_{10},\ q_{15}\}$
$RS_6^{ff3} = \{q_0,\ q_1,\ q_7,\ q_8,\ q_9,\ q_{15}\}$
$RS_7^{ff3} = \{q_0,\ q_7,\ q_8,\ q_{15}\}$
$RS_8^{ff3} = \{q_7,\ q_{15}\}$
$RS_9^{ff3} = \{\emptyset\}$

Finally, let us consider Sets CS_r of states where flip-flop r has just changed. Sets CS_r are as follows:

$CS_{ff0} = \{q_0,\ q_1,\ q_2,\ q_3,\ q_4,\ q_5,\ q_6,\ q_7,\ q_8,\ q_9,\ q_{10},\ q_{11},\ q_{12},\ q_{13},\ q_{14},\ q_{15}\}$
$CS_{ff1} = \{q_1,\ q_3,\ q_5,\ q_7,\ q_9,\ q_{11},\ q_{13},\ q_{15}\}$
$CS_{ff2} = \{q_3,\ q_7,\ q_{11},\ q_{15}\}$
$CS_{ff3} = \{q_7,\ q_{15}\}$

These RS^r_k and CS_r are used in timing analysis of paths between flip-flops for computing the maximum allowable clock cycle of paths detailed in Section 2.5.

2.5. Multi-Cycle Path Detection

In this section, we show a timing analysis method of each path between flip-flops.

2.5.1 Interval of Value Changes

In general, combinational parts of sequential logic circuits have many paths between flip-flops as shown in Figure 2.5, and the maximum allowable clock cycle of each path may be different. We would like to know the clock cycles between the input flip-flop change and the output flip-flop change. That may vary on each state and the minimum cycle of allowable clocks decides the clock period. Formally, the following property is satisfied.

Property 2.2 (Interval of value change) Let *in* and *out* be flip-flops, the interval of value change from *in* to *out* is k, when the following condition is satisfied:

$$CS_{in} \subseteq RS^{out}_k$$

□

2.5.2 Interval Analysis of Value Change

Let *in* and *out* be flip-flop, and there be a path from *in* to *out*. From the Property 2.2, if $CS_{in} \subseteq RS^{out}_k$, then for $i < k$ $CS_{in} \subseteq RS^{out}_i$. Hence, the maximum allowable clock cycle of the path from *in* to *out* is the maximum number of i satisfying $CS_{in} \subseteq RS^{out}_i$. The maximum number of i is computed as the procedure

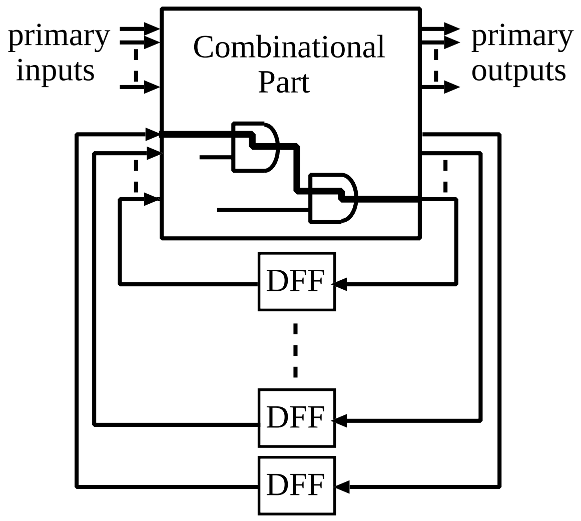


Figure 2.5. Paths between flip-flops.

shown in Figure 2.6, where i is incremented from 2 to $\mathcal{K}_{out}$ while $CS_{in} \not\subseteq RS_i^{out}$. If i is equal to $\mathcal{K}_{out}$, then the allowable clock cycles of the path is infinite. Note that allowable clock cycles of a path becomes infinite when the destination flip-flop of the path never changes.

If the maximum allowable clock cycle of some paths is k (≥ 2), then the path becomes false with respect to the clock, and the allowable delay time of the path is clock period multiplied by k.

The analysis procedure is as follows.

(P1) Compute the reachable state set RS.

(P2) For each flip-flop r, compute CS_r.

(P3) For each flip-flop r, compute RS_k^r. for $k = 2, 3, ...$

(P4) For each flip-flop pair (r, s), compute Interval (r, s) as shown in Figure 2.6.

Let n be the number of flip-flops, and *#repetition* be the number of repetitions in computing the reachable state set. The computation of RS in P1 is that of the usual symbolic state traversal based on BDD, and the computation time is proportional to the *#repetition*. The computation of P2 is proportional to n, and each CS_r can be obtained with one BDD operation on RS. The computation of

```
Interval($in$, $out$)
  $in$, $out$: the flip-flops which have paths from $in$ to $out$
  $CS_{in}$: the state set where the value of $in$ changes
  $RS_i^{out}$: the state set where the value of $out$ does not
       change during $i$ clock cycles
  $i$: the interval of the value change between $in$ and $out$
begin
  $i$ = 2;
  while ($CS_{in} \subseteq RS_i^{out}$) do
    if ($i == \mathcal{K}_{out}$) then
      return $\infty$;
    end if
    $i$ = $i$ + 1;
  end while
  return $i$ - 1;
end
```

Figure 2.6. Analysis of the interval of value change between flip-flops.

P3 is within $n \times \#repetition$ steps, since k of RS_k^r is bounded by the $\#repetition$. The computation of P4 is within $n^2 \times \#repetition$ steps.

[Example]

To illustrate timing analysis of paths between flip-flops, let consider a 4-bit counter shown in Figure 2.1 and results of update cycle analysis shown in Section 2.4. There are 6 flip-flop pairs which have a path in the circuit, that are (ff0, ff1), (ff1, ff2), (ff0, ff2), (ff2, ff3), (ff1, ff3) and (ff0, ff3).

Let us consider the maximum allowable clock cycle of these paths which is computed using results(sets of states) of update cycle analysis. The results of the maximum allowable clock cycle detection are shown in Table 2.1. In the table, “Path” shows the path between flip-flops, and “Clock cycles” shows the maximum allowable clock cycle of the path. For instance, the maximum allowable clock cycle of the path from ff0 to ff1 is 1, and that of the path from ff2 to ff3 is

Table 2.1. Results for 4-bit synchronous counter.

Path	Clock cycles
ff0 $\longrightarrow$ ff1	1
ff1 $\longrightarrow$ ff2	2
ff0 $\longrightarrow$ ff2	1
ff2 $\longrightarrow$ ff3	4
ff1 $\longrightarrow$ ff3	2
ff0 $\longrightarrow$ ff3	1

4.

For the former, the interval of value change from ff0 to ff1 is 1 because of $CS_{ff0} \not\subseteq RS_2^{ff1}$, and the maximum allowable cycles of the path ff0 $\rightarrow$ ff1 is 1. For the latter, the interval of value change from ff2 to ff3 is 4 because of $CS_{ff2} \subseteq RS_2^{ff3}$, $CS_{ff2} \subseteq RS_3^{ff3}$, $CS_{ff2} \subseteq RS_4^{ff3}$ and $CS_{ff2} \not\subseteq RS_5^{ff3}$. Thus the maximum allowable cycles of the path ff2 $\rightarrow$ ff3 is 4. Similarly, the maximum allowable clock cycle of the others (ff1 $\rightarrow$ ff2, ff0 $\rightarrow$ ff2, ff1 $\rightarrow$ ff3 and ff0 $\rightarrow$ ff3) are computed.

2.6. Experimental Results

We have implemented the ideas proposed in this chapter in C language. We have applied the method to a prime number generator and the ISCAS89 benchmarks [4] on a Sun Ultra2 workstation (CPU UltraSPARC II 300MHz, Main memory 512MB).

2.6.1 Results on Prime Number Generator

A prime number generator is a circuit which computes all prime numbers ($\leq$ 250) starting from 2. Figure 2.7 shows the algorithm we used for computing prime numbers. The algorithm sequentially checks all odd numbers ($\leq$ 250). Each candidate number "is_prime" is divided by odd numbers "t_reg1" less than "is_prime"/2 in turn to decide whether the number is prime or not. Division is implemented with iteration of addition/subtraction. "t_reg2" is used for computing division. We have designed the prime number generator in VHDL. The VHDL design is non-pipeline and includes hardware loop structures. We obtained a gate level circuit from the VHDL design with the Synopsys Design Compiler[1], where the circuit consists of 35 flip-flops and about 500 gates.

We have applied our method to the circuit and examined 772 flip-flop pairs which have paths between them. We have found 476 flip-flop pairs with allowable clock cycles greater than 1. 190 pairs among them can be guessed from the state transitions in the VHDL design. The others (286 pairs) are hard to guess from VHDL description. 133 pairs of the 190 pairs relate to counters controlling the hardware loop. The others of 190 pairs come from the structure of the circuit, in which the value changes are controlled by wait states as shown in the Section 3. The number of repetitions in computing reachable states of the circuit was 11959. The elapsed CPU time was 3619.0 seconds on the Sun Ultra2.

We have checked critical paths in the prime number circuit, where the delay is measured as the number of gates on the path. We have found 16 critical paths (43 gates), and 14 of them have been detected by our method as paths whose allowable clock cycles are more than 2.

These results show that many multi-cycle paths exist in sequential logic circuits.

[1]The Synopsys Design Compiler is licensed in the education program of VLSI Design and Education Center (VDEC), the University of Tokyo. I would like to thank to VDEC (VLSI Design and Education Center) for the support of Synopsys tools.

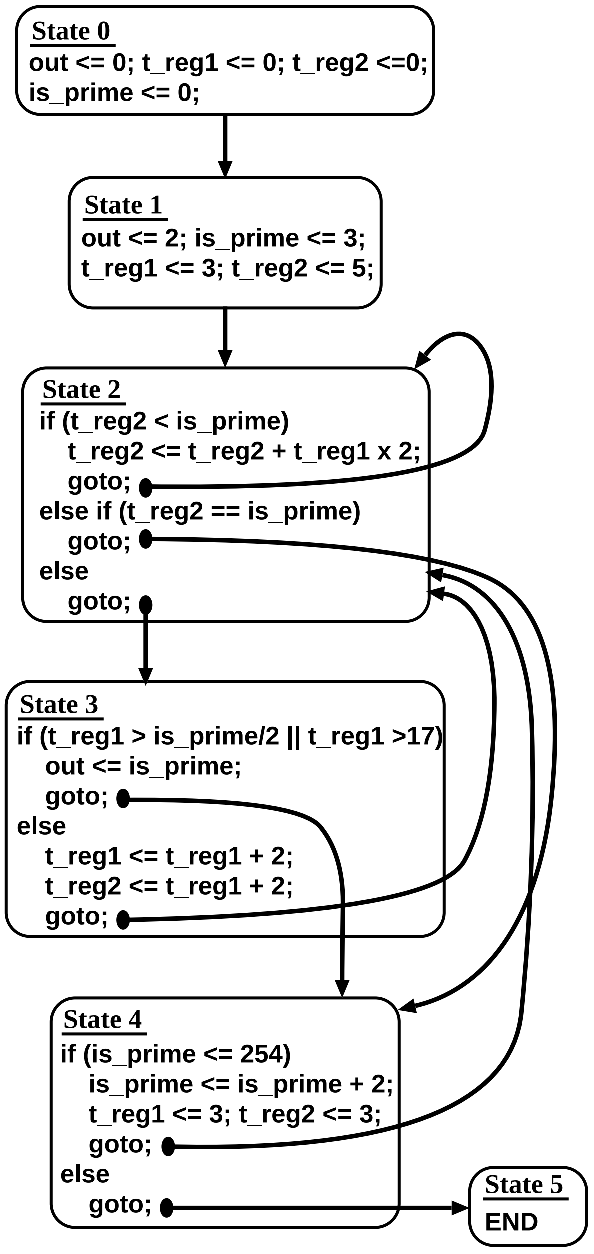


Figure 2.7. Algorithm for computing prime numbers.

2.6.2 Results on ISCAS89 Benchmarks

We have applied our detection method to 32 ISCAS89 benchmarks [4] and found multi-cycle flip-flop pairs with allowable clock cycles greater than 1 on 22 circuits. Table 2.2 shows the statistics. In the table, "#In","#FF" and "#FF-pair" are the number of primary inputs, flip-flops and connected pairs of flip-flops in the circuit respectively. "#Rep" is the number of repetitions in computing reachable states of the circuit. "#M-pair" is the number of flip-flop pairs whose maximum allowable clock cycles are 2 or more.

The prime number generator and s838 include almost the same number of flip-flops, but the elapsed CPU time of s838 was quite short compared with the prime number generator. This is because #Rep of the prime number generator is 11959 and that of s838 is 17. Each repetition includes BDD manipulations, and the execution time heavily depends on the #Rep.

We have checked critical paths in 30 benchmarks. The critical paths whose allowable clock cycles are 2 or more have been found in 11 circuits. In Table 2.2, † denotes that the circuit includes such critical paths.

Table 2.2. Result of multi-cycle path analysis of ISCAS benchmark circuits.

Circuits				Multi-Cycle Paths Analysis		
Name	#In	#FF	#FF-pair	#Rep	#M-pair	Time [sec]
daio	1	4	6	5	0	3.0
ex1†	10	20	380	10	357	3.0
ex2†	2	19	342	6	306	3.0
ex3†	2	10	90	5	80	3.0
ex4†	6	14	169	14	135	3.0
ex5†	2	9	72	4	48	3.0
ex6†	5	9	61	1	61	3.0
ex7	2	10	90	5	77	3.0
s27	4	3	4	3	0	2.0
s208†	11	8	28	17	18	3.0
s298	3	14	56	19	4	4.0
s344	9	15	74	7	1	79.0
s349	9	15	74	7	1	50.0
s382	3	21	131	151	13	150.0
s386	7	6	30	8	4	3.0
s420†	19	16	72	17	62	3.0
s444	3	21	131	151	13	112.0
s510	19	6	30	47	7	3.0
s526	3	21	123	151	8	204.0
s526n	3	21	123	151	8	225.0
s641†	35	19	100	7	38	55.0
s713†	35	19	100	7	38	55.0
s820	18	5	20	11	0	3.0
s832	18	5	20	11	0	3.0
s838†	35	32	160	17	150	4.0
s953	16	29	150	11	29	279.0
s1196	14	18	20	3	0	802.0
s1238	14	18	20	3	0	1074.0
s1488	8	6	30	23	0	3.0
s1494	8	6	30	23	0	3.0

2.7. Conclusion

In this chapter, we have proposed a method to detect the multi-cycle paths. Multi-cycle paths exist when input and output flip-flops of the paths are guarded with wait states, and the delay time of the path can be greater than the clock period. More precisely, we should decide the clock period under the constraint where the delay of the path is smaller than the clock period multiplied by the maximum allowable clock cycle.

In the detection of multi-cycle paths, we use the symbolic state traversal to obtain the state sets with update cycle more than 2. We have applied the detection method to ISCAS benchmarks and found multi-cycle paths on 22 circuits in 30 finite state machine benchmarks. Since our method includes BDD manipulations, the applicable circuit size is restricted by the memory size for manipulating BDD's. At present technology, circuits including 50-100 flip-flops can be manipulated. For applying large circuits, we should consider (1) circuit partitioning: a method to partition a large circuit to independent small modules for which our method can be applied, (2) analysis on VHDL sources: a method to detect multi-cycle paths on VHDL source and to omit the gate level analysis for such paths, and (3) abstraction of datapath: a method to reduce datapath width of some flip-flop pairs which can be analyzed on VHDL sources.

The method proposed in this chapter can be applied to decide the maximum clock frequency of sequential logic circuits and to optimize the delay of some operations in logic synthesis systems.

Chapter 3

Multi-Cycle Path Analysis Based on Propositional Satisfiability

3.1. Introduction

Two multi-cycle path detection methods [10, 16] have been proposed. One is based on the analysis of state transition graphs of the controllers of microprocessors [10], and the other is based on the symbolic state traversal of finite state machines (FSMs) [16]. Both methods are based on the state traversal of finite state machines, and for large circuits these suffer from the huge size of reachable states.

Propositional satisfiability (SAT) is a method to decide whether a Boolean formula is satisfiable or not, and has been used in logic comparison and test pattern generation [13]. Recently, SAT is applied to the analysis of sequential circuits, such as symbolic model checking [3] and timing analysis of combinational circuits [20]. SAT is also used to the critical delay path analysis [20] of combinational logic circuits.

In this chapter, we present a SAT-based multi-cycle path detection method. The method generates a conjunctive normal form (CNF) formula which is satisfiable if and only if the path is not a multi-cycle path, and checks the satisfiability of the formula with a SAT prover. To reduce the size of CNF formulae, we also introduce heuristics on the conversion from multi-level circuits into CNF formulae. The heuristics adaptively insert intermediate variables with considering the structure of logic networks in the conversion, and reduce the size of CNF formulae. The set of reachable states is hard to manipulate based on SAT based-methods, so we adopt an approximation that all states are reachable. By this approximation, some multi-cycle paths may not be detected, but there are many detectable multi-cycle paths in large circuits under the approximation as shown in the chapter. Note that such paths cannot be detected without the approximation, and that the information of the detected paths is effective to estimate or optimize the timing issues of the parts including the paths.

We have implemented our method and tested it on ISCAS89 benchmarks [4] and other sample circuits. Experimental results show the improvement in the size of manipulatable circuits, especially hard circuits for BDD manipulation.

This chapter is organized as follows. In the next section, we present Propositional Satisfiability. In Section 3.3, we define multi-cycle paths and show a detection method of multi-cycle paths. In Section 3.4, we present a satisfiability based multi-cycle path detection method. In Section 3.5, we present heuristics on the conversion from multi-level circuits into CNF formulae. In Section 3.6, we show the experimental results.

3.2. Propositional Satisfiability

Let x_i be a propositional variable, and $\overline{x}_i$ be the negation of x_i. A variable and its negation are called *literals*. The disjunction (OR, $\vee$) of *literals* such as $x_1 \vee x_2$ and $x_1 \vee \overline{x}_2 \vee x_3$ is a *clause*. The conjunction of *clauses* such as $(x_1 \vee x_2) \wedge (x_1 \vee \overline{x}_2 \vee x_3)$ is a conjunctive normal form (CNF) formula. A CNF formula f over $x_1, ..., x_l$ is a CNF formula including only *literals* $x_1, \overline{x}_1, x_2, \overline{x}_2, ..., x_l, \overline{x}_l$.

Let $(x_1, ..., x_l)$ be a tuple of propositional variables and f be a CNF formula over $x_1, ..., x_l$. We can assign T(true) or F(false) for x_i. If x_i is assigned T(F), then $\overline{x}_i$ is assigned F(T). A *clause* has value T, if T is assigned for one of literals in the clause. A CNF formula has value T, if all *clauses* have value T. A CNF formula f is *satisfiable* if f has value T with some value assignments for variables.

3.3. Multi-Cycle Path Detection Based on Symbolic State Traversal of FSM

Given a circuit, multi-cycle paths are detected by analyzing FSM model of the circuit as described in Chapter 2. At first, we compute the set RS of reachable states from the initial state, and we analyze properties of FSM on RS.

Let r_{in} and r_{out} be flip-flops, and there be a path from r_{in} to r_{out}. If the value of r_{out} does not change at the next clock of the clock when the value of r_{in} has just changed, then the propagation of signals can use 2 or more clock cycles. Note that the change of r_{in} does not contribute to the value of r_{out}. We obtain the following condition specifying the multi-cycle property on a path from r_{in} to r_{out}:

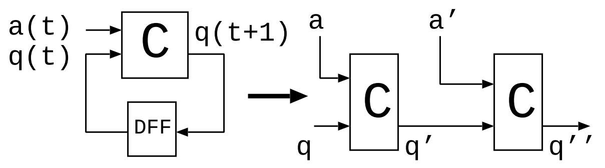
$$\begin{aligned} &\forall q \in RS,\ \forall a, a' \in \Sigma, \\ &\Big[(q' = \delta(q,\ a)) \wedge (q'' = \delta(q',\ a')) \\ &\quad \rightarrow \Big((q(r_{in}) \neq q'(r_{in})) \rightarrow (q'(r_{out}) = q''(r_{out})) \Big) \Big]. \end{aligned} \tag{1}$$

Formula (1) denotes that if the value of r_{in} changes at the state transition from q to q', then the value of r_{out} does not change at the state transition from q' to q'', where q, q' and q'' are reachable from the initial state and $q(r)$ denotes the value of a flip-flop r under the state q. Similarly, the condition specifying the path can use 3 or more clock cycles can be obtained by extending Formula (1).

In the following, we focus on Formula (1) specifying the allowable clock cycles to be 2 or more. At present, the symbolic state traversal of the FSM is the best way to compute RS, and a BDD-based multi-cycle path detection method has been proposed [16]. However, the method is hard to apply to large circuits. Hence, we relax the above condition a little and check the relaxed condition using propositional satisfiability.

3.4. Multi-Cycle Path Detection Using Satisfiability

In this section, we describe an algorithm that detects multi-cycle paths based on propositional satisfiability.



(a) A sequential circuit (b) A time expansion

Figure 3.1. Time expansion of a sequential circuit.

SAT is not suitable for computing the reachable state set. Thus we abandon the computation and assume that all states are reachable from the initial state. The following formula is used in the multi-cycle path analysis instead of Formula (1).

$$\begin{aligned}&\forall q \in S,\ \forall a, a' \in \Sigma,\\ &\Big[(q' = \delta(q,\ a)) \wedge (q'' = \delta(q',\ a')) \\ &\quad \rightarrow \Big((q(r_{in}) \neq q'(r_{in})) \rightarrow (q'(r_{out}) = q''(r_{out})) \Big) \Big].\end{aligned} \tag{2}$$

Since Formula (2) takes into account all states, there is a possibility that several multi-cycle paths can not be detected by the analysis as we described in 2.3. However, we should adopt some approximation to manipulate large circuits. Experimental results show the effect of the approximation. The accuracy of the analysis is reduced. The information of detected multi-cycle paths, however, is the same as that with the exact analysis, and is effective to estimate or optimize the parts including the paths.

3.4.1 Time Expansion

The analysis starts from the time expansion of sequential circuits, where a sequential circuit is translated into a multi-level combinational circuit. Figure 3.1 shows the idea of the time expansion: (a) is a sequential circuit and (b) is a time expanded combinational circuit. Since we consider Formula (2), the duplication of a circuit as shown in Figure 3.1(b) is enough. Symbols in Figure 3.1(b) correspond to those in Formula (2), where "C" implements the state transition function δ.

3.4.2 Propositional Formula for Multi-Cycle Path Analysis

Based on Formula (2), we define a propositional formula F for detecting multi-cycle paths.

Let r_{in} and r_{out} be the input and output flip-flops to be checked, and let $\equiv$ and $\not\equiv$ denote equivalence and negation of equivalence between propositional variables respectively. The following propositional formula F becomes true if the path from r_{in} to r_{out} can use 2 clock cycles:

$$\begin{aligned} F \overset{\triangle}{=} & T(a,\ q,\ q') \wedge T(a',\ q',\ q'') \\ & \qquad \rightarrow \Big((q(r_{in}) \not\equiv q'(r_{in})) \rightarrow (q'(r_{out}) \equiv q''(r_{out}))\Big), \end{aligned} \tag{3}$$

where $T(a,\ q,\ q') = 1$ iff $q' = \delta(q, a)$ and $T(a',\ q',\ q'') = 1$ iff $q'' = \delta(q', a')$. Note that $T(a,\ q,\ q')$ represents a logic formula which is implemented by a multi-level circuit, where a, q and q' are coded in binary and represented by a tuple of propositional variables. Also note that $q(r_{in})$ and $q(r_{out})$ are propositional variables corresponding to r_{in} and r_{out} in the tuple of the state variables for the state q.

If $\overline{F}$ is unsatisfiable, then F is always true for any q, a and a', and therefore we can decide that the allowable clock cycle is 2 or more. We convert $\overline{F}$ into CNF formulae, and check the unsatisfiability of $\overline{F}$ using a SAT prover. We can obtain CNF of $\overline{F}$ using a conversion rule $a \not\equiv b \Rightarrow (a \vee b) \wedge (\overline{a} \vee \overline{b})$:

$$\begin{aligned} \overline{F} = & T(a,\ q,\ q') \wedge T(a',\ q',\ q'') \\ & \wedge (q(r_{in}) \vee q'(r_{in})) \wedge (\overline{q(r_{in})} \vee \overline{q'(r_{in})}) \\ & \wedge (q'(r_{out}) \vee q''(r_{out})) \wedge (\overline{q'(r_{out})} \vee \overline{q''(r_{out})}). \end{aligned} \tag{4}$$

Note that $T(a,\ q,\ q')$ and $T(a',\ q',\ q'')$ are the same logic formula with different variables. In the following we discuss a method to generate CNF of $T(a,\ q,\ q')$.

3.5. Conversion to CNF Formula

In this section we show a method to convert multi-level combinational circuits into CNF formulae and heuristics on the conversion.

3.5.1 Conversion to CNF Formula without New Variables

In general, CNF formulae can be generated from multi-level combinational circuits using the following conversion rules: $(f_1 \not\equiv f_2) \Rightarrow (f_1 \vee f_2) \wedge (\overline{f_1} \vee \overline{f_2})$, $(f_1 \equiv f_2) \Rightarrow (f_1 \vee \overline{f_2}) \wedge (\overline{f_1} \vee f_2)$ and $(f_1 \wedge f_2) \vee f_3 \Rightarrow (f_1 \vee f_3) \wedge (f_2 \vee f_3)$. The size of resulting CNF formulae may grow exponentially, and we should introduce heuristic techniques on the conversion. For example, let f_1, f_2, f_3 and f_4 be subformulae which are not CNF, and we consider a formula $(f_1 \wedge f_2) \vee (f_3 \wedge f_4)$. The formula is converted into $(f_1 \vee f_3) \wedge (f_1 \vee f_4) \wedge (f_2 \vee f_3) \wedge (f_2 \vee f_4)$, and the size of the formula can be twice. If f_3 and f_4 are $f_{31} \wedge f_{32}$ and $f_{41} \wedge f_{42}$ respectively, then f_1 and f_2 should be duplicated in the conversion to CNF and the size of CNF can be exponential with respect to the size of the original formula in the worst case. As above, $\vee$ causes exponential growth of formulae in the conversion to CNF. In addition to $\vee$, $\not\equiv$ and $\equiv$ cause exponential growth of formulae because both $\not\equiv$ and $\equiv$ are converted into two $\vee$ using the above rules. In the following, we assume that multi-level combinational logic circuits consist of AND, OR, NOT and XOR ($\not\equiv$) gates.

3.5.2 Conversion to CNF Formula with New Variables

To avoid the problem in the conversion to CNF, we apply a method which inserts new propositional variables in the conversion and reduces the size of CNF formulae without affecting the satisfiability of formulae [18].

For example, let a_0, a_1, b_0, b_1 and c_0 be propositional variables, and we consider a formula $a_1 \not\equiv b_1 \not\equiv ((a_0 \wedge b_0) \vee ((a_0 \not\equiv b_0) \wedge c_0))$. If we insert new variables x_1, x_2, x_3, x_4 and x_5 for $a_1 \not\equiv b_1$, $x_3 \vee x_4$, $a_0 \wedge b_0$, $x_5 \wedge c_0$ and $a_0 \not\equiv b_0$ respectively, then we obtain an equivalent formula as follows: $(x_1 \not\equiv x_2) \wedge (x_1 \equiv (a_1 \not\equiv b_1)) \wedge (x_2 \equiv (x_3 \vee x_4)) \wedge (x_3 \equiv (a_0 \wedge b_0)) \wedge (x_4 \equiv (x_5 \wedge c_0)) \wedge (x_5 \equiv (a_0 \not\equiv b_0))$. From the equivalent formula, we obtain the following CNF formula applying the above rules: $(\overline{x5} \vee \overline{a0} \vee \overline{b0}) \wedge (\overline{x5} \vee a0 \vee b0) \wedge (x5 \vee b0 \vee \overline{a0}) \wedge (x5 \vee a0 \vee \overline{b0}) \wedge (x4 \vee \overline{x5} \vee \overline{c0}) \wedge (\overline{x4} \vee c0) \wedge (\overline{x4} \vee x5) \wedge (x3 \vee \overline{a0} \vee \overline{b0}) \wedge (\overline{x3} \vee a0) \wedge (\overline{x3} \vee b0) \wedge (\overline{x2} \vee x3 \vee x4) \wedge (x2 \vee \overline{x4}) \wedge (x2 \vee \overline{x3}) \wedge (\overline{x1} \vee \overline{a1} \vee \overline{b1}) \wedge (\overline{x1} \vee a1 \vee b1) \wedge (x1 \vee b1 \vee \overline{a1}) \wedge (x1 \vee a1 \vee \overline{b1}) \wedge (\overline{x1} \vee \overline{x2}) \wedge (x1 \vee x2)$, and the number of literals of the formula is 49. New variables are useful for reducing the size of CNF formulae and the number of literals.

On the other hand, if we do not insert new variables, we obtain the following CNF formula applying the above conversion rules: $(\overline{a0}\vee\overline{a1}\vee\overline{b0}\vee b1)\wedge(\overline{a0}\vee a1\vee\overline{b0}\vee\overline{b1})\wedge(\overline{a0}\vee\overline{a1}\vee b0\vee b1\vee\overline{c0})\wedge(\overline{a0}\vee a1\vee b0\vee\overline{b1}\vee\overline{c0})\wedge(a0\vee\overline{a1}\vee\overline{b0}\vee b1\vee\overline{c0})\wedge(a0\vee a1\vee\overline{b0}\vee\overline{b1}\vee\overline{c0})\wedge(a0\vee\overline{a1}\vee\overline{b1}\vee c0)\wedge(a0\vee a1\vee b1\vee c0)\wedge(\overline{a1}\vee b0\vee\overline{b1}\vee c0)\wedge(a1\vee b0\vee b1\vee c0)\wedge(a0\vee\overline{a1}\vee b0\vee\overline{b1})\wedge(a1\vee a0\vee b0\vee b1)\wedge(a0\vee\overline{a1}\vee b0\vee\overline{b1})\wedge(a0\vee a1\vee b0\vee b1)$, and the number of literals of the formula is 60. If we introduce variables for any logic operation with 2 operands, we can easily show that the size of CNF is less than 12 $\times$ (the number of operations) as shown in [18].

3.5.3 Adaptive Variable Insertion

In [18], new variables are inserted for all logic gates in the conversion. For many circuits, the method is not best on the size of generated formulae. For example, we consider $a_1 \not\equiv b_1 \not\equiv ((a_0 \wedge b_0) \vee ((a_0 \not\equiv b_0)) \wedge c_0)$ again. If we insert only two new variables x_1 and x_2 for $a_1 \not\equiv b_1$ and $(a_0 \wedge b_0) \vee ((a_0 \not\equiv b_0) \wedge c_0)$, then we obtain an equivalent formula as follows: $(x_1 \not\equiv x_2) \wedge (x_1 \equiv (a_1 \not\equiv b_1)) \wedge (x_2 \equiv ((a_0 \wedge b_0) \vee ((a_0 \not\equiv b_0) \wedge c_0)))$. From the equivalent formula, we obtain the following CNF formula applying the above conversion rules: $(\overline{x2}\vee a0\vee c0)\wedge(\overline{x2}\vee b0\vee c0)\wedge(\overline{x2}\vee a0\vee b0)\wedge(\overline{x2}\vee a0\vee b0)\wedge(x2\vee\overline{a0}\vee\overline{b0})\wedge(x2\vee\overline{a0}\vee b0\vee\overline{c0})\wedge(x2\vee a0\vee\overline{b0}\vee\overline{c0})\wedge(\overline{x1}\vee\overline{a1}\vee\overline{b1})\wedge(\overline{x1}\vee a1\vee b1)\wedge(x1\vee\overline{a1}\vee b1)\wedge(x1\vee a1\vee\overline{b1})\wedge(\overline{x1}\vee\overline{x2})\wedge(x1\vee x2)$, and the number of literals of the formula is 39. The size of CNF formulae is reduced by selective variable insertion. We should control the number of new variables and reduce the size of CNF formulae because the execution time of SAT provers is affected by the size of formulae.

In the variable insertion, we should treat carefully XOR and OR gates because XOR and OR gates cause exponential growth of CNF formulae as we described in Section 3.5.1. For AND and NOT gates, we need not introduce variables, since they are suitable to convert CNF formulae. Hence, we use threshold values for controlling the variable insertion, that are sensitive to the logic type of a gate and the depth of logic gates. Our method adaptively inserts new variables based on the structure of logic networks, namely, the depth of logic gates from primary output and that from newly inserted variables are considered. Figure 3.2 shows the idea. We classify 4 cases as shown in the figure, and introduce 4 parameters on the level. In Figure 3.2, "v" denotes a logic gate which we focus on, "Var"-s denote new variables which we inserted. “XOR-level”, “OR-level”, “AND-level” and “NOT-level” are thresholds that denote the maximum depth from newly inserted variables. We insert new variables at the focusing gate “v” when the number of levels of “v” from the nearest variables exceeds the thresholds. The circuit is divided into sub-circuits with almost the same level. For example, the OR gate in Figure 3.2(b) is considered. If the level of the OR gate from the nearest variable is over “OR-level” and “pre_v” is not a NOT gate, then we introduce new variables for fan-ins of the OR gate. Note that we do not insert a new variable if “pre_v” is a NOT gate because NOT-OR corresponds to an AND gate in the conversion to CNF. Similarly, we insert a new variable for NOT-AND as in Figure 3.2(c) because NOT-AND corresponds to an OR gate that causes the exponential growth of formulae. We show the effect of this method on reducing CNF size in Section 6.

Figure 3.3 shows an algorithm for inserting new variables. The analysis starts from the primary outputs of multi-level logic circuits, and goes backward toward primary inputs based on the depth first search algorithm. At first, *lev* is set to 1, and then *lev* is incremented for every recursive calls of `DepthFirst()`. If the value of *lev* exceeds the threshold depending the type of each gate, then new variables are inserted for fan-ins by `NewVar`().

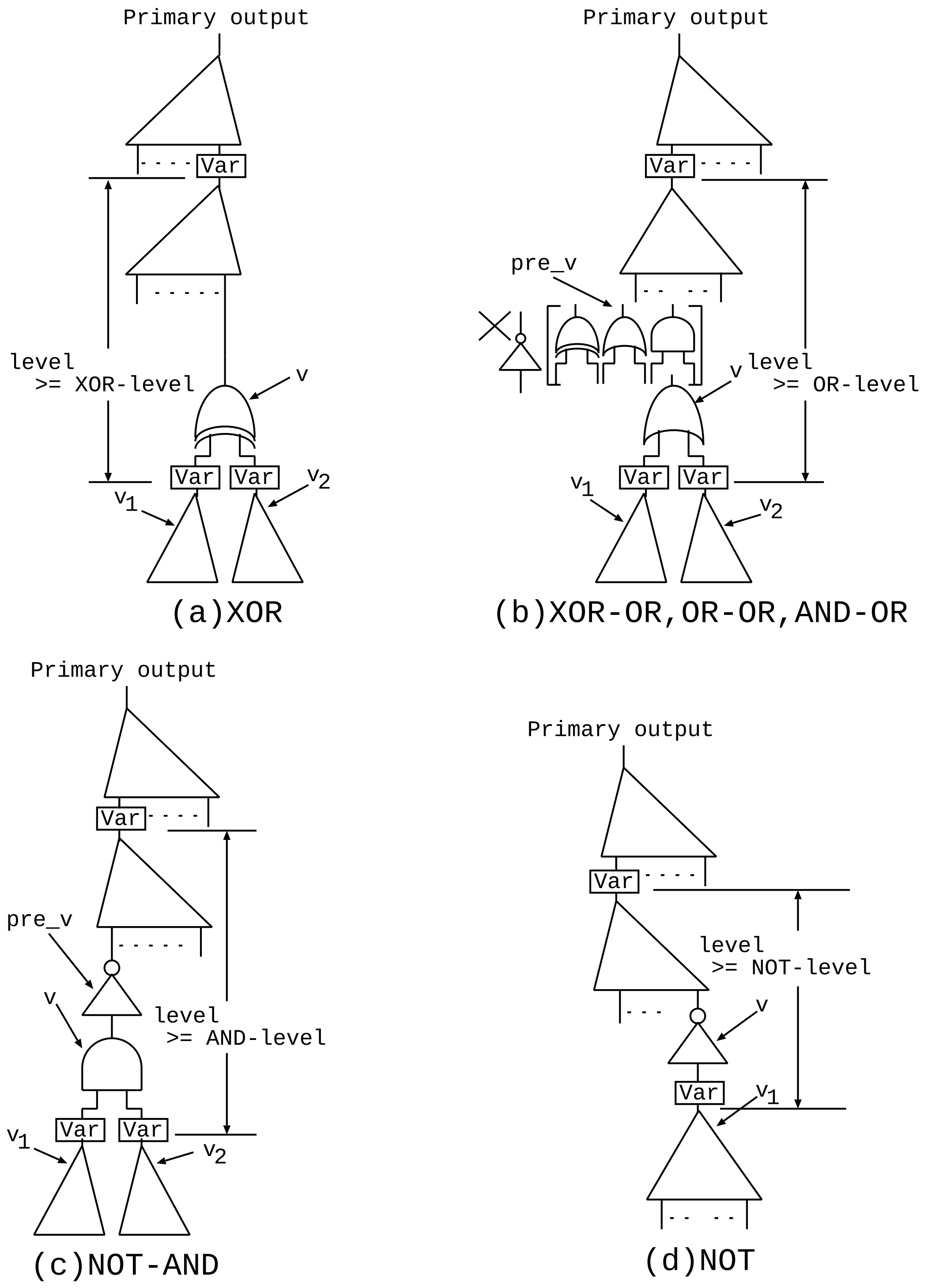


Figure 3.2. Internal variables.

```
DepthFirst ($v$,$pre\_v$,$lev$)
  $v$:  a gate;
  $pre\_v$:  the previous gate of $v$;
  $lev$:  a level from $v$ to variable;
  $v_1, v_2$:  the next gate of $v$;
  XOR-level, OR-level, AND-level, NOT-level:  thresholds;
  Var:  primary inputs/flip-flops of the circuit;
  NewlyInsertedVar:  new variables inserted by NewVar();
begin
  switch ($v$)
    case ($v$ == Var):  return;
    case ($v$ == NewlyInsertedVar):  return;
    case ($v$ == XOR): /*Figure 3.2(a)*/
      if (XOR-level <= $lev$)
        NewVar($v_1$); NewVar($v_2$); $lev$ = 1;
      else
        $lev$ = $lev$ + 1;
      DepthFirst($v_1$,$v$,$lev$); DepthFirst($v_2$,$v$,$lev$);
      return;
    case ($v$ == OR and $pre\_v \neq$ NOT): /*Figure 3.2(b)*/
      if (OR-level <= $lev$)
        NewVar($v_1$); NewVar($v_2$); $lev$ = 1;
      else
        $lev$ = $lev$ + 1;
      DepthFirst($v_1$,$v$,$lev$); DepthFirst ($v_2$,$v$,$lev$);
      return;
    case ($v$ == AND and $pre\_v$ == NOT): /*Figure3.2(c)*/
      if (AND-level <= $lev$)
        NewVar($v_1$); NewVar($v_2$); $lev$ = 1;
      else
        $lev$ = $lev$ + 1;
      DepthFirst($v_1$,$v$,$lev$); DepthFirst($v_2$,$v$,$lev$);
      return;
    case ($v$ == NOT): /* Figure 3.2(d) */
      if (NOT-level <= $lev$)
        NewVar($v_1$); $lev$ = 1;
      else
        $lev$ = $lev$ + 1;
      DepthFirst ($v_1$,$v$,$lev$);
      return;
  end switch;
end
```

Figure 3.3. An algorithm for introducing new variables.

3.6. Experimental Results

We have implemented a SAT-based multi-cycle path analyzer in C language and compared it with a multi-cycle path analyzer based on symbolic state traversal of FSMs using BDD's [16]. We have also implemented BDD-based analyzer without reachability. The SAT-based analyzer reads circuit descriptions in SLIF [21], and produces CNF formulae in DIMACS format [12] for detecting multi-cycle paths. The satisfiability of the formula is checked with H. Zhang's SATO [22] / J. Silva's GRASP [19]. Both of the SAT provers are based on Davis-Putnam method [8].

We have analyzed the ISCAS89 benchmarks [4] and other sample circuits designed in our laboratory on a PC (CPU Pentium II 500MHz, Main memory 512MB). In the experiment, we use the threshold (1, 4, 4, 5) for (XOR-level, OR-level, AND-level, NOT-level) of the algorithm in Figure 3.3.

3.6.1 Results on ISCAS Benchmarks

We have applied 3 analysis methods to 44 ISCAS89 benchmarks. (1) Method1: BDD-based algorithm which takes into account the reachable state set [16]. (2) Method2: BDD-based algorithm which assumes that all states are reachable, (3) Method3: SAT-based algorithm which assumes that all states are reachable. We used SATO as a SAT prover.

Table 3.1 and 3.2 show the statistics. In the tables, "#In", "#FF" and "#FF-pair" are the number of primary inputs, flip-flops and connected pairs of flip-flops in the circuit respectively. "#Rep", "#M-pair" and "Time" are the number of state traversal repetitions to compute reachable state set of the circuit, the number of pairs of flip-flops whose path is a multi-cycle path(2-cycle), and the elapsed CPU seconds obtained by the time command respectively. "#Var", "#Cl" and "#Lit" are the maximum number of propositional variables, clauses and literals of the formula which is used in our method. Multi-cycle paths on 21 circuits are found by Method1 [16]: ex2, ex3, ex4, ex5, ex6, ex7, s208, s298, s344, s349, s382, s386, s420, s444, s510, s526, s526n, s641, s713, s838, s953. Multi-cycle paths on only 16 circuits among them can be found by Method2 and Method3: ex2, ex3, ex4, ex5, ex6, ex7, s298, s344, s349, s382, s386, s444, s510, s526, s526n, s953. Because of the effect of unreachable states, multi-cycle paths in s208, s420,

Table 3.1. Result of multi-cycle path analysis of ISCAS89 benchmark circuits(1).

Circuits				Method1 (Reachability, BDD)			Method2 (BDD)		Method3 (SAT)				
Name	#In	#FF	#FF-pair	#Rep	#M-pair	Time	#M-pair	Time	#M-pair	#Var	#Cl	#Lit	Time
daio	1	4	6	5	0	0.8	0	2.8	0	18	46	124	0.7
ex2	2	19	342	6	306	1.0	238	2.1	238	195	1250	3554	4.6
ex3	2	10	90	5	80	0.8	45	1.7	45	102	570	1626	1.2
ex4	6	14	169	14	135	0.9	135	1.7	135	116	436	1172	1.5
ex5	2	9	72	4	48	0.8	30	1.5	30	93	488	1380	1.0
ex6	5	9	61	1	61	0.5	9	1.6	9	99	638	2032	0.9
ex7	2	10	90	5	77	0.8	53	1.8	53	102	614	1816	1.2
s27	4	3	4	3	0	0.6	0	1.5	0	23	48	124	0.5
s208	11	8	28	17	18	0.9	0	1.6	0	78	240	700	0.5
s298	3	14	56	19	4	0.7	3	1.7	3	112	468	1438	0.9
s344	9	15	74	7	1	1.9	1	1.7	1	155	480	1296	1.3
s349	9	15	74	7	1	1.8	1	1.6	1	155	484	1312	1.3
s382	3	21	131	151	13	11.6	13	9.9	13	175	622	1738	2.0
s386	7	6	30	8	4	0.9	4	1.5	4	78	500	1824	0.9
s420	19	16	72	17	62	0.9	0	1.7	0	158	496	1420	1.1
s444	3	21	131	151	13	13.3	13	6.3	13	221	844	2392	2.6
s510	19	6	30	47	7	0.9	2	1.6	2	192	814	2588	1.4
s526	3	21	123	151	8	10.3	7	20.1	7	207	1030	3344	2.5
s526n	3	21	123	151	8	10.5	7	37.7	7	203	1024	3338	2.5
s641	35	19	100	7	38	9.3	0	5.3	0	259	656	1818	1.9
s713	35	19	100	7	38	10.2	0	6.0	0	293	844	2344	2.6
s820	18	5	20	11	0	0.9	0	1.5	0	221	1504	5768	1.7
s832	18	5	20	11	0	0.8	0	1.7	0	217	1538	5902	1.6
s838	35	32	160	17	150	1.1	0	101.1	0	318	1008	2860	3.0
s953	16	29	150	11	29	19.9	29	6.2	29	469	1702	4848	6.2
s1196	14	18	20	3	0	253.4	0	10.9	0	318	1290	3856	2.2
s1238	14	18	20	3	0	200.2	0	12.6	0	322	1338	4030	2.4
s1423	17	74	1694	—	Mem > 1G	—	Mem > 1G	—	46	654	2238	6298	60.4
s1488	8	6	30	23	0	0.9	0	1.6	0	222	1576	5836	2.0
s1494	8	6	30	23	0	0.8	0	1.7	0	222	1608	5980	2.0
Total number of M-pair				1101			590		636				

Mem > 1G: we have analyzed on a DEC AlphaServer8400 (CPU alpha21164A 617MHz $\times$ 10, Main memory 8GB).

Table 3.2. Result of multi-cycle path analysis of ISCAS89 benchmark circuits(2).

Circuits				Method1 (Reachability, BDD)			Method2 (BDD)		Method3 (SAT)				
Name	#In	#FF	#FF-pair	#Rep	#M-pair	Time	#M-pair	Time	#M-pair	#Var	#Cl	#Lit	Time
prolog	36	136	551	—	Mem > 1G	—	—	Mem > 1G	29	1080	5490	19878	60.9
s1512	29	57	456	—	Time > 48h	—	—	Mem > 1G	1	555	2058	6006	13.2
s3271	26	116	789	—	Mem > 1G	—	—	Mem > 1G	0	1384	5292	15574	53.4
s3330	40	132	522	—	Mem > 1G	—	—	Mem > 1G	0	1080	12876	67086	126.6
s3384	43	183	1759	—	Mem > 1G	—	—	Mem > 1G	0	1757	5724	17044	128.8
s4863	49	104	628	—	Mem > 1G	—	—	Mem > 1G	0	2054	9420	29284	76.7
s6669	83	239	2155	—	Mem > 1G	—	—	Mem > 1G	51	2713	11584	35928	323.6
s9234	19	228	2680	—	Mem > 1G	—	—	Mem > 1G	150	3114	11206	33370	407.5
s13207	31	699	3406	—	Mem > 1G	—	—	Mem > 1G	797	5875	17586	49148	911.2
s15850.1	77	534	11497	—	Mem > 1G	—	—	Mem > 1G	237	6196	21378	60766	3288.5
s35932	35	1728	4475	—	Mem > 1G	—	—	Mem > 1G	0	19810	61490	159932	7661.7
s38417	28	1636	32774	—	Mem > 1G	—	—	Mem > 1G	206	16034	56798	184252	29099.6
s38584	12	1452	16880	—	Time > 48h	—	—	Mem > 1G	1595	12864	56862	182544	15260.4
s38584.1	38	1426	15300	—	Mem > 1G	—	—	Mem > 1G	15	12778	56724	182548	14348.7
Total number of M-pair				0			0		3081				

Mem $>$ 1G: we have analyzed on a DEC AlphaServer8400 (CPU alpha21164A 617MHz $\times$ 10, Main memory 8GB).

Time: we have measureded on a PC (CPU Pentium III 750MHz, Main memory 512MB).

s641, s713 and s838 can not be found by Method2 and Method3, and the total number of paths is small on Method2 and Method3 for detected circuits. The total number of "#M-pair" with Method 1 is 1101, and those with Method 2 and 3 are 590 and 3717. Table 3.1 shows that more than half of M-pairs can be detected under the assumption where all states are reachable. Note that Method 3 can detect some of M-pairs of s1423, prolog, s1512, s6669, s9234, s13207, s15850.1, s38417, s38584 and s38584.1 which include the node explosion of BDDs.

3.6.2 Results on Sample Circuits

We have applied our SAT-based method and the BDD-based method in [16] to several circuits which are designed in our laboratory: prime250 (a prime number generator which computes prime numbers less than 250, about 400 gates), s-div8 (an 8-bit sequential divider, about 500 gates), prime999 (a prime number generator which computes prime numbers less than 999, about 400 gates), bezier8 (an 8-bit bezier curve generator, about 900 gates), forsen (an edge detection circuit for image processing, about 900 gates), pcpu16 (a 16-bit pipelined processor, about 9,000 gates), mul64 (a circuit including a 64-bit combinational adder array multiplier, about 47,000 gates), SpchRecog (an HMM-based speech recognition circuit, about 40,000 gates) [17], EyeTracking (an eye tracking circuit, about 45,000 gates) [15]. These circuits have been designed to include multi-cycle paths. Multi-cycle paths of forsen are introduced to meet the clock constraints, those of SpchRecog and EyeTracking are introduced for memory access constraints. Multi-cycle paths of mul64 are introduced for experiments. We have found multi-cycle flip-flop pairs on all circuits except pcpu16 by Method3. pcpu16 may not include any multi-cycle flip-flop pairs since the circuit consists of pipelined modules. Table 3.3 shows the statistics. Almost all circuits, BDD-based methods suffer from memory overflow or time-overflow. We used SATO as a SAT prover if not specified. For SpchRecog and EyeTracking, we should use GRASP to overcome the number of flip-flops more than 1000.

Most of multi-cycle flip-flop pairs detected by the analysis relate to counters controlling the hardware loop and the structure of the circuit, in which the value changes are controlled by wait states.

Table 3.3. Results of multi-cycle path analysis of sample circuits.

Circuits				Method1 (Reachability, BDD)			Method2 (BDD)	
Name	#In	#FF	#FF-pair	#Rep	#M-pair	Time	#M-pair	Time
prime250	2	35	718	11959	570	774.3	413	116606.6
s-div8	10	47	514	—	Mem>1G	—	—	Time>48h
prime999	2	44	1219	—	Mem>1G	—	—	Time>48h
bezier8	34	64	684	—	Mem>1G	—	Mem>1G	—
forsen	187	255	1739	—	Mem>1G	—	Mem>1G	—
pcpu16	40	779	6744	—	Mem>1G	—	Mem>1G	—
mul64	259	258	12866	—	Mem>1G	—	Mem>1G	—
SpchRecog	37	1316	134443	—	Mem>1G	—	Mem>1G	—
EyeTracking	32	2163	171529	—	Mem>1G	—	Mem>1G	—

Circuits				Method3 (SAT)				
Name	#In	#FF	#FF-pair	#M-pair	#Var	#Cl	#Lit	Time
prime250	2	35	718	413	435	1662	4768	17.4
s-div8	10	47	514	33	557	2000	5620	20.5
prime999	2	44	1219	480	550	2258	6498	37.1
bezier8	34	64	684	58	2121	9074	27264	37.1
forsen	187	255	1739	1739	2547	956	24922	199.9
pcpu16	40	779	6744	0	4675	22462	71762	16292.9
mul64	259	258	12866	12480	39666	160114	451304	97560.6
SpchRecog	37	1316	134443	57573	22256	105252	328272	547.5h†
EyeTracking	32	2163	171529	121999	26369	139652	500968	579.3h†

Mem>1G: we have analyzed on a DEC AlphaServer8400 (CPU alpha21164A 617MHz × 10, Main memory 8GB).

SpchRecog, EyeTracking: we have analyzed on 6 Sun Ultra2 workstations (UltraSPARC-IIi 333MHz, Main memory 512MB) in parallel.

†: The total CPU time using GRASP.

3.6.3 Effects of Thresholds for Inserting New Variables

We have tested properties of thresholds which are used to insert new variables in our algorithm. We have experimented with two medium size circuits and two large circuits in Table 3.1 and Table 3.3.

Table 3.4 shows statistics of multi-cycle path detection of s1423, mul16, mul64 and SpchRecog. In the table, “Threshold: a, b, c, d” denote “XOR-level”, “OR-level”, “AND-level” and “NOT-level” in Figure 3.2. Since XOR gates grow the size of CNF formulae heavily as we described in Section 3.5.1, “XOR-level” is set to 1. Note that our algorithm with (1, 1, 1, 2) inserts variables for all logic gates.

From these tables, we can see the effect of the threshold values on the size of CNF formulae and the elapsed CPU time. The statistics show that the reduction of the size of CNF formula does not always contribute to reduce the time for analysis. The analysis with thresholds (1, 9, 9, 10) need the shortest CPU time for medium size circuits, but the analysis need much time or memory than that with (1, 4, 4, 5) for large circuits. The analysis with (1, 4, 4, 5) finished in all cases with reasonable time and memory.

Table 3.4. The influence of the threshold on multi-cycle path analysis.

Threshold	Method3 (SAT)			
a, b, c ,d	#Var	#Cl	#Lit	Time
1, 14, 14, 15	414	4604	25222	95.7
1, 9, 9, 10	460	2248	7896	47.5
1, 4, 4, 5	654	2238	6290	57.8
1, 1, 1, 2	1256	3232	7500	83.1

(a) Statistics for s1423

Threshold	Method3 (SAT)			
a, b, c ,d	#Var	#Cl	#Lit	Time
1, 14, 14, 15	2324	44034	440910	799.8
1, 9, 9, 10	2398	11164	35570	138.6
1, 4, 4, 5	2824	10890	30800	179.4
1, 1, 1, 2	4714	14084	35168	369.3

(b) Statistics for mul16

Threshold	Method3 (SAT)			
a, b, c ,d	#Var	#Cl	#Lit	Time
1, 14, 14, 15	—	—	—	Mem>1G†
1, 9, 9, 10	35234	165422	516944	198685.9
1, 4, 4, 5	39666	160114	451304	97560.6
1, 1, 1, 2	—	—	—	Mem>1G†

(c) Statistics for mul64

Threshold	Method3 (SAT)			
a, b, c ,d	#Var	#Cl	#Lit	Time
1, 14, 14, 15	—	—	—	Mem>1G‡
1, 9, 9, 10	17140	2894616	51223668	Mem>1G‡‡
1, 4, 4, 5	22256	105252	328272	547.5h†
1, 1, 1, 2	37302	126864	327198	Time>24h††

(d) Statistics for SpchRecog

Mem>1G: we have analyzed on a DEC AlphaServer8400 (CPU alpha21164A 617MHz × 10, Main memory 8GB).

†: the total CPU time using GRASP on 6 Sun Ultra2 workstations (UltraSPARC-IIi 333MHz, Main memory 512MB) in parallel. The average execution time of SAT prover for one formula is 15[sec].

††: time over flow of SAT prover on a Sun Ultra2 workstations (UltraSPARC-IIi 333MHz, Main memory 512MB). The execution time of SAT prover for one formula > 24[h].

‡: memory over flow on CNF generation.

‡‡: memory over flow of SAT prover.

3.7. Conclusion

In this chapter, we have presented multi-cycle path detection method based on propositional satisfiability and shown experimental results.

The method reduces multi-cycle path detection problems into SAT problems. In the conversion from multi-level circuits into CNF formulae, the method adaptively inserts intermediate variables to reduce the size of CNF formulae which are given to a SAT prover.

The SAT-based algorithm enables us to apply multi-cycle path analysis to large circuits that can not be analyzed with the symbolic state traversal based algorithm.

We have applied our method to ISCAS89 benchmarks and other sample circuits. Experimental results show the improvement on the manipulatable size of circuits by using satisfiability.

The problem is that the SAT-based algorithm can only detect a subset of multi-cycle paths detectable by the algorithm based on the symbolic state traversal of FSMs. We should develop SAT-based manipulations of reachable states.

Chapter 4

Application of Multi-Cycle Path Analysis to Logic Synthesis

4.1. Area Optimization with Information of Multi-Cycle Paths

The number of allowable clock cycles for each path is computed in the multi-cycle paths detection, and the number is useful for optimizing circuits in logic synthesis. We describe the area optimization of circuits with the information of allowable clock cycles of paths.

In general, the area of circuits becomes small as the delay time of circuits becomes long. This is trade-off between the area and the delay time of circuits. Hence, when a circuit includes many multi-cycle paths the area of the circuit may be reduced. Circuits which include multi-cycle paths are optimized by the following process.

1. Multi-cycle path detection:
   we detect multi-cycle paths with the method proposed in this thesis.

2. Multi-cycle path specification:
   we specify the multi-cycle paths for the logic synthesis.

3. Multi-cycle path constrained synthesis:
   we optimize the circuit under the timing constraints which are based on

multi-cycle paths.

[Example]

In Synopsys Design Compiler, multi-cycle paths can be handled and can be specified with "set_multi_cycle_path" command. We show an example of the area optimization of circuits with the information of multi-cycle paths. We have designed two circuits (add8 and add16) which execute addition with 2 clock cycles. add8 includes one 8-bit adder and two 8-bit registers(A and B), and computes A = A + B. Similarly, add16 includes one 16-bit adder and two 16-bit registers. We have optimized the circuits in area for generating small circuits, where the maximum delay times of add8 and add16 are constrained as 15 ns and 30 ns respectively. We used Synopsis Design Compiler and lsi_10k library. Table 4.1 shows the statistics. In the table, "clock frequency" is the constraint of clock frequency, "non-Multi" is the combinational area (total cell area) of the circuit which is yielded by the area constrained synthesis, "Multi" is the combinational area (total cell area) of the circuit which is yielded by the area and multi-cycle paths constrained synthesis and "reduction ratio" is the reduction ratio in area that is (non-Multi − Multi)/non-Multi. These results show that the area of circuits can be reduced with the information of multi-cycle paths: 26 % in combinational area, 7 % in total cell area for add8 and add16.

Table 4.1. The area of circuits.

Circuit	clock frequency	non-Multi	Multi	reduction ratio
add8	66 MHz	135 (368)	101 (343)	0.26 (0.07)
add16	33 MHz	266 (695)	197 (647)	0.26 (0.07)

We can also find redundant flip-flops whose values never change, since allowable clock cycles of a path becomes infinite when the destination flip-flop of the path never changes. The information of such flip-flops can be used to reduce the area of circuits by changing such flip-flops to logic values.

4.2. Experimental Results

We have optimized two prime number generators (prime250 and prime100) based on the example in Section 4.1. prime250 and prime100 are circuits which compute prime numbers less than 250 and 100 respectively. The maximum delay time of the circuits is constrained as 20 ns. Table 4.2 shows the statistics. In the table, each column shows the same item in Table 4.1. These results show that synthesis of circuits with the information of multi-cycle paths yields a slightly smaller circuit: 5 % in combinational area, 3 % in total cell area for prime250 and 9 % in combinational area, 5 % in total cell area for prime100.

Table 4.2. The area of prime number generators.

Circuit	clock frequency	non-Multi	Multi	reduction ratio
prime250	50 MHz	375 (836)	358 (810)	0.05 (0.03)
prime100	50 MHz	333 (739)	302 (700)	0.09 (0.05)

Chapter 5

Conclusion

In this thesis, we have formalized multi-cycle paths and proposed two detection methods of multi-cycle paths, one is based on a symbolic state traversal of FSMs and the other is based on propositional satisfiability.

In Chapter 2, we have proposed a method to detect the multi-cycle paths. Multi-cycle paths exist when input and output flip-flops of the paths are guarded with wait states, and the delay time of the path can be greater than the clock period. Hence, we should decide the clock period under the constraint where the delay of the path is smaller than the clock period multiplied by the maximum allowable clock cycle. In the detection of multi-cycle paths, we use the symbolic state traversal to obtain the state sets with update cycle more than 2. We have applied the detection method to ISCAS benchmarks and found multi-cycle paths on 22 circuits in 30 finite state machine benchmarks. Multi-cycle path analysis can be applied to maximize the clock frequency of sequential logic circuits. Since our method includes BDD manipulations, the applicable circuit size is restricted by the memory size for manipulating BDD's. At present technology, circuits including 50-100 flip-flops can be manipulated by the method.

In Chapter 3, we have presented multi-cycle path detection method based on propositional satisfiability and shown experimental results. The method reduces multi-cycle path detection problems into SAT problems. In the conversion from multi-level circuits into CNF formulae, the method adaptively inserts intermediate variables to reduce the size of CNF formulae which are given to a SAT prover. The SAT-based algorithm enables us to apply multi-cycle path analysis to large circuits that can not be analyzed with the algorithm based on the symbolic state traversal of FSMs. We have applied our method to ISCAS89 benchmarks and other sample circuits. Experimental results show the improvement on the manipulatable size of circuits by using satisfiability. The problem is that the SAT-based algorithm can only detect a subset of multi-cycle paths detectable by the symbolic execution based algorithm. We should develop SAT-based manipulations of reachable states.

In Chapter 4, we have shown logic optimization with the information of detected multi-cycle paths. The informations on multi-cycle paths can be used in logic optimization to reduce the number of gates and the size of gates with considering allowable clock cycles of each path.

Multi-cycle path analysis is important for the correct timing analysis, and the multi-cycle path detection method proposed in this thesis contributes to improve the accuracy of timing analysis. However, in sequential circuits, there still exist paths which are sensitizable but do not affect the clock period. For example, if the value of a flip-flop r_{out} is not referred at some clock cycle, then the signal to r_{out} may not be propagated at the cycle. In other words, there may be a 3-cycle path including an intermediate flip-flop whose value is not referred in some clock cycle. For the precise timing analysis, we should develop manipulation methods of such paths.

Acknowledgements

I would like to express my sincere appreciation to Professor Katsumasa Watanabe of Nara Institute of Science and Technology for his continuous guidance, helpful suggestions, accurate criticisms and encouragement for this research.

I would also like to appreciate Professor Hideo Fujiwara and Professor Akira Fukuda of Nara Institute of Science and Technology for their invaluable comments and helpful suggestions concerning this thesis.

I would also like to express my gratitude to Associate Professor Shinji Kimura of Nara Institute of Science and Technology who introduced me to this research field, and has been giving me invaluable suggestions, accurate criticisms and encouragement through this research.

I would also like to express my thanks to Lecturer Kazuyoshi Takagi of Nagoya University for his valuable discussions and comments.

I would also like to thank Research Associate Takashi Horiyama and Research Associate Masaki Nakanishi of Nara Institute of Science and Technology for their discussions and helpful support throughout this research.

I also wish to express my thanks to Associate Professor Satoshi Yamane of Kagoshima University for his helpful suggestions and encouragement when I was an under graduate student of Shimane University.

I also wish to thank Mr. Shinji Maruoka of Hitachi, Ltd. for his helpful discussions when he was a master course student of Nara Institute of Science and Technology.

Thanks are due to all members of Watanabe Laboratory for their discussions and helpful comments.

Lastly, I thank my parents for their patience, support and encouragement.

References

[1] P. Ashar, S. Dey, and S. Malik. “Exploiting Multicycle False Paths in the Performance Optimization of Sequential Logic Circuits”. *IEEE Transactions on CAD*, vol-14(9), pp. 1067–1075, Sep. 1995.

[2] J. Benkowski, E.V. Meersch, L.J.M. Claesen, and H. DE MAN. “Timing Verification Using Statistically Sensitizing Path”. *IEEE Transactions on CAD*, vol-9(10), pp. 1073–1084, Oct. 1990.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. “Symbolic Model Checking using SAT procedures instead of BDDs”. In *of 36th Design Automation Conference (DAC'99)*, 1999.

[4] F. Brglez, D. Bryan, and K. Kozminski. “Combinational Profiles of Sequential Benchmark Circuits”. In *IEEE 1989 International Symposium on Circuits and Systems Proceedings*, pp. 1929–1934, May 1989.

[5] R.E. Bryant. “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams”. *ACM Computing Surveys*, vol.24(3), pp. 293–318, Sep. 1992.

[6] H.C. Chen and D.H. Du. “Path Sensitization in Critical Path Problem”. In *Proc. of International Conference on Computer Aided Design (ICCAD'91)*, pp. 208–221, 1991.

[7] O. Coudert, C. Berthet, and J. C Madre. “Verification of Sequential Machines Based on symbolic Execution”. In *LNCS 407, Automatic Verification Methods for Finite State Systems*, pp. 365–373, Sep. 1989.

[8] M. Davis and H. Putnam. "A Computing Procedure for Quantification Theory". *Journal of the Association for Computing Machinery*, 7, pp.201–215, 1960.

[9] M. Fujita and E. M. Clarke. "Application of BDD to CAD for digital systems". *Information Processing*, Vol. 34(5), pp. 609–616, May 1993.

[10] Anurag P. Gupta and Daniel P. Siewiorek. "Automated Multi-Cycle Symbolic Timing Verification of Microprocessor-based Designs". In *Proc. of 31st Design Automation Conference (DAC'94)*, pp. 113–119, 1994.

[11] J.E. Hopcroft and J.D. Ullman. *"Introduction to Automata Theory, Languages and Computation"*. Addison-Wesley, 1979.

[12] D. S. Johnson and editors M. A. Trick. "The second DIMACS implementation challenge". In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1993, (see `http://dimacs.rutgers.edu/Challenges/`).

[13] Tracy Larrabee. "Test Pattern Generation Using Boolean Satisfiability". *IEEE Transaction on Computer-Aided Design*, pp. 4–15, Jan. 1992.

[14] P. C. McGeer, A. Saldanha, R. K. Brayton, and A. Sangiovanni-Vincentelli. "Delay Models and Exact Timing Analysis". In *Logic Synthesis and Optimization, Kluwer Academic Publishers*, pp. 167–189, 1993, ed. T. Sasao.

[15] Kazuhiro Nakamura, Masaki Nakanishi, Takashi Horiyama, Masato Suzuhi, Shinji Kimura, and Katsumasa Watanabe. "Real-time Eye Tracking LSI for Eye-based Interface". In *Proc. of 4th System LSI Biwako Workshop*, pp. 267–270, 2000.

[16] Kazuhiro Nakamura, Kazuyoshi Takagi, Shinji Kimura, and Katsumasa Watanabe. "Waiting False Path Analysis of Sequential Logic Circuits for Performance Optimization". In *Proc. of International Conference on Computer Aided Design (ICCAD'98)*, pp. 392-395, 1998.

[17] Kazuhiro Nakamura, Qiang Zhu, Shinji Maruoka, Takashi Horiyama, Shinji Kimura, and Katsumasa Watanabe. "Speech Recognition Chip for Monosyl-

lables". In *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC 2001)*, pp. 396-399, 2001.

[18] D. Plaisted and S. Greenbaum. "A Structure-Preserving Clause Form Translation". *Journal of Symbolic Computation*, 2, pp. 293–304, 1986.

[19] J. Silva and K. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability". *IEEE Transactions on Computers*, pp. 506–521, May 1999.

[20] L. G. Silva, J. P. M. Silva, L. M. Silveira, and K. A. Sakallah. "Timing Analysis Using Propositional Satisfiability". In *IEEE International Conference on Electronics, Circuits and Systems*, 1998.

[21] S. Yang. "Logic Synthesis and Optimization Benchmarks User Guide". In *Technical Report 1991-IWLS-UG-Saeyang*, 1991.

[22] H. Zhang. "SATO: An Efficient Propositional Prover". In *of International Conference on Automated Deduction'97*, pp. 272–275, 1997.

List of Publications

Journal Papers

1. Kazuhiro Nakamura, Shinji Kimura, Kazuyoshi Takagi and Katsumasa Watanabe, "Timing Verification of Sequential Logic Circuits Based on Controlled Multi-clock Path Analysis," IEICE Transactions on Fundamentals, Vol. E81-A, No. 12, pp. 2515-2520, December 1998.

2. Kazuhiro Nakamura, Shinji Maruoka, Shinji Kimura and Katsumasa Watanabe, "Multi-Cycle Path Detection Based on Propositional Satisfiability with CNF Simplification Using Adaptive Variable Insertion," IEICE Transactions on Fundamentals, Vol. E83-A, No. 12, pp. 2600-2607, December 2000.

International Conferences

1. Kazuhiro Nakamura, Kazuyoshi Takagi, Shinji Kimura and Katsumasa Watanabe, "Waiting False Path Analysis of Sequential Logic Circuits for Performance Optimization," Proceedings of IEEE/ ACM International Conference on Computer Aided Design 1998, pp. 392-395, November 1998.

2. Kazuhiro Nakamura, Shinji Maruoka, Shinji Kimura and Katsumasa Watanabe, "Multi-Clock Path Analysis Using Propositional Satisfiability," Proceedings of Asia and South Pacific Design Automation Conference 2000, pp. 81-86, January 2000.

3. Kazuhiro Nakamura, Qiang Zhu, Shinji Maruoka, Takashi Horiyama, Shinji Kimura and Katsumasa Watanabe, "Speech Recognition Chip for Monosyllables," Proceedings of Asia and South Pacific Design Automation Conference 2001, pp. 396-399, January 2001.

4. Kazuhiro Nakamura, Qiang Zhu, Shinji Maruoka, Takashi Horiyama, Shinji Kimura and Katsumasa Watanabe, "A Real-time 64-Monosyllable Recognition LSI with Learning Mechanism," Proceedings of Asia and South Pacific Design Automation Conference 2001, pp. 31-32, January 2001.

Technical Reports

1. Kazuhiro Nakamura, Shinji Kimura, Kazuyoshi Takagi and Katsumasa Watanabe, “Waiting False Path Analysis of Sequential Logic Circuits,” Technical Report of IEICE, VLD97-132, ICD97-237, pp. 71-77, March 1998, (in Japanese).

2. Kazuhiro Nakamura, Shinji Maruoka, Shinji Kimura and Katsumasa Watanabe, “Multi-Clock Path Analysis Based on Propositional Satisfiability,” Technical Report of IEICE, VLD99-82, ICD99-211, FTS99-60, pp. 55-62, November 1999, (in Japanese).

3. Kazuhiro Nakamura, Shinji Maruoka, Shinji Kimura and Katsumasa Watanabe, “State Enumeration Based on Propositional Satisfiability,” Technical Report of IEICE, VLD2000-26, CAS200-17,DSP2000-38, pp. 123-130, June 2000, (in Japanese).

4. Kazuhiro Nakamura, Qiang Zhu, Shinji Maruoka, Takashi Horiyama, Shinji Kimura and Katsumasa Watanabe, “Design and Implementation of LSI for Speech Recognition with Learning Mechanism Using C Language,” Technical Report of IEICE, VLD2000-90, ICD200-147, FTS2000-55, pp. 125-130, November 2000, (in Japanese).

5. Kazuhiro Nakamura, Masaki Nakanishi, Takashi Horiyama, Masato Suzuhi, Shinji Kimura and Katsumasa Watanabe, “Real-time Eye Tracking LSI for Eye-based Interface,” 4th System LSI Biwako Workshop, pp. 267-270, November 2000, (in Japanese).

6. Kazuhiro Nakamura, Shinji Kimura, Kazuyoshi Takagi and Katsumasa Watanabe, “Timing Verification Using Waiting False Path Analysis of General Conference of IEICE, A-3-8, pp. 94, March 1998, (in Japanese).

7. Shinji Maruoka, Kazuhiro Nakamura, Shinji Kimura and Katsumasa Watanabe, “Propositional Satisfiability for Multi-Level Logic Circuits and its Application to Multi-Clock Path Analysis,” Society Conference of IEICE, A-3-8, pp. 51, September 1999, (in Japanese).

8. Kazuhiro Nakamura, Qiang Zhu, Shinji Maruoka, Takashi Horiyama, Shinji Kimura and Katsumasa Watanabe, "Speaker Independent Speech Recognition Circuit with Learning Circuit Interface," Society Conference of IEICE, A-3-13, pp. 80, October 2000, (in Japanese).

Other Publications

International Conference

1. Kazuhiro Nakamura and Satoshi Yamane, "Formal Verification of Real-Time Software by Symbolic Model Checker," Proceedings of International Conference on Application of Concurrency to System Design 1998, pp. 99-108, March 1998.

Technical Reports

1. Satoshi Yamane and Kazuhiro Nakamura, "Symbolic Model-Checking Verification of Timed Automata Based on Region Graph Method," Technical Report of IEICE, FTS96-28, pp. 1-8, June 1996, (in Japanese).

2. Kazuhiro Nakamura and Satoshi Yamane, "Real-time Symbolic Model-Checking Method Based on Approximations by BDD and DBM," 10th Karuizawa Workshop on Circuits and Systems, pp. 77-82, April 1997, (in Japanese).

3. Kenichi Takahashi, Kazuhiro Nakamura and Satoshi Yamane, "Formal Verification of Speed-Independent Circuit Based on BDD," General Conference of IEICE, D-10-2, pp. 231, March 1997, (in Japanese).

4. Kazuhiro Nakamura and Satoshi Yamane, "Real-time Symbolic Model-Checking Method Based on Approximations by BDD and DBM," General Conference of IEICE, D-10-3, pp. 232, March 1997, (in Japanese).