ABSTRACT

TIMING VERIFICATION AND OPTIMIZATION OF CIRCUITS WITH LEVEL-SENSITIVE LATCHES

by

Timothy Michael Burks

Chair: Karem Sakallah

This work describes the development of a model for the static timing analysis of circuits with level-sensitive latches and applies it to three major timing analysis problems: timing verification, optimal clocking, and timing-driven design. The model describes both level-sensitive latches and edge-triggered flip-flops; it is used to accurately verify circuit timing and to demonstrate that the properties of level-sensitive latches can be exploited to improve circuit performance. An extended formulation of the Critical Path Method (CPM) is developed which incorporates the timing model for level-sensitive latches. The types of critical paths which can arise within this formulation are discussed, analyzed, and related to existing procedures for timing verification of circuits with level-sensitive latches. After describing the optimal clocking problem, we present methods for finding solutions for several simplified cases and present an extended formulation that allows branch-and-bound and MILP solutions for the general problem, allowing the first exact solutions of unrestricted optimal clocking problems. Finally, we explore the application of common critical-path based optimizations to the extended CPM formulation for level-sensitive latches. Two problems are studied: the problem of timing- and area-driven part selection and the problem of ordering inputs of CMOS gates to minimize cycle time. For each problem, we show that optimization results can be significantly improved by using the extended CPM formulation.
TIMING VERIFICATION AND OPTIMIZATION
OF CIRCUITS WITH LEVEL-SENSITIVE LATCHES

by

Timothy Michael Burks

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1994

Doctoral Committee:

Associate Professor Karem A. Sakallah, Chair
Professor Edward S. Davidson
Professor Trevor N. Mudge
Professor Katta G. Murty
Thomas G. Szymanski, Head, Computer Systems Research, AT&T Bell Laboratories
ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support and encouragement of many people, many of whom can never be adequately thanked for their help. First of all, I must thank my wife, Linda, for her constant friendship and encouragement and for keeping our lives together during my last few months of writing.

Thanks also to Matthew Jones and many other friends and colleagues: Joao-Paulo Silva, John-David Wellman, Mike Riepe, Chuan-Hua Chang, Wei-Han Lien, Emily Shriver, Ayman Kayssi and others who have shared long discussions, pizza, technical assistance, and advice. Ayman Kayssi also supplied the FrameMaker document formats for this dissertation.

A special thanks goes to Karem Sakallah, who served as my academic advisor and provided the initial core of ideas upon which this thesis is based. His suggestions, reviews, and encouragement at various stages of my research are gratefully acknowledged. I am also grateful to each of the other members of my dissertation committee: Ed Davidson, Trevor Mudge, Katta Murty, and Tom Szymanski. Each was as valuable for their wisdom and encouragement as for their technical advice; I am proud to have been able to work with each of them.

The majority of my graduate studies was generously supported by the U. S. Department of Army’s National Defense Science and Engineering Graduate Fellowship. Their support is gratefully acknowledged.

Finally, I’d like to thank my parents for the years of education that mattered most and my son, Michael, for the years to come.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

I. INTRODUCTION 1

1.1 Timing Analysis of Synchronous Digital Circuits 1
1.2 Existing Work on Timing Analysis 6
  1.2.1 Timing Verification 6
  1.2.2 Optimal Clocking 18
  1.2.3 Timing-Driven Design 20
  1.2.4 The False Path Problem 25
1.3 Thesis Organization 26

II. TIMING MODELS AND DEFINITIONS 28

2.1 Timing Modeling 28
2.2 A Timing Model for Synchronous Circuits 29
  2.2.1 Model Parameters 30
  2.2.2 Model Constraints 34
  2.2.3 Graph Representation of Constraints 40
2.3 Gate-Level Extension of the Timing Model 44
2.4 Modeling Issues 45
  2.4.1 Clock Skew Effects 45
  2.4.2 Latch Operating Modes 47
  2.4.3 Wave Pipelining 49
  2.4.4 Stoppable and Restartable Machines 54
  2.4.5 Inputs, Outputs, and Hierarchical Models 55
2.5 Conclusions 58

III. TIMING VERIFICATION 59

3.1 The Timing Verification Problem 59
3.2 A Simple Verification Algorithm 60
3.3 Critical Paths in Circuits with Level-Sensitive Latches 63
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Synchronizer Types</td>
<td>32</td>
</tr>
<tr>
<td>2.2</td>
<td>Timing Model Summary</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Timing Model Expressed in a Global Frame-of-Reference</td>
<td>100</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of mmLP Solution Methods</td>
<td>121</td>
</tr>
<tr>
<td>4.3</td>
<td>Optimal Clocking of Single-Phase Level-Sensitive ISCAS89 Benchmarks</td>
<td>123</td>
</tr>
<tr>
<td>5.1</td>
<td>Test Circuit Details</td>
<td>130</td>
</tr>
<tr>
<td>5.2</td>
<td>SIZE-TILOS results at $T_H$</td>
<td>154</td>
</tr>
<tr>
<td>5.3</td>
<td>SIZE-TILOS results at $T_L$</td>
<td>155</td>
</tr>
<tr>
<td>5.4</td>
<td>SIZE-HINSBERGER-KOLLA results at $T_H$</td>
<td>156</td>
</tr>
<tr>
<td>5.5</td>
<td>SIZE-HINSBERGER-KOLLA results at $T_L$</td>
<td>157</td>
</tr>
<tr>
<td>5.6</td>
<td>SIZE-UP results at $T_H$</td>
<td>158</td>
</tr>
<tr>
<td>5.7</td>
<td>SIZE-UP results at $T_L$</td>
<td>159</td>
</tr>
<tr>
<td>5.8</td>
<td>SIZE-DOWN results at $T_H$</td>
<td>160</td>
</tr>
<tr>
<td>5.9</td>
<td>SIZE-DOWN results at $T_L$</td>
<td>161</td>
</tr>
<tr>
<td>5.10</td>
<td>Attainable Minimum Cycle Times for Various Net Loading Factors</td>
<td>161</td>
</tr>
<tr>
<td>5.11</td>
<td>Sizing Results for s13207 with Various Net Loading Factors</td>
<td>162</td>
</tr>
<tr>
<td>5.12</td>
<td>Input Sorting Results</td>
<td>170</td>
</tr>
<tr>
<td>5.13</td>
<td>Input Sorting Results (after sizing to equalize delays)</td>
<td>171</td>
</tr>
<tr>
<td>5.14</td>
<td>Input Sorting Results for t953 (after sizing) with Various Delay Factors</td>
<td>171</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Digital System Design Levels</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Level-Sensitive Latches vs. Edge-Triggered Flip-Flops</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Operation of Circuit with Level-Sensitive Latches</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Sample CPM Project Network (Arrow Diagram)</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>Types of Float in CPM Project Networks</td>
<td>9</td>
</tr>
<tr>
<td>1.6</td>
<td>AoN Formulation of the CPM Example</td>
<td>11</td>
</tr>
<tr>
<td>1.7</td>
<td>Alternate Formulations of the Circuit Timing Verification Problem</td>
<td>13</td>
</tr>
<tr>
<td>1.8</td>
<td>Thesis Structure</td>
<td>27</td>
</tr>
<tr>
<td>2.1</td>
<td>Circuit Model</td>
<td>29</td>
</tr>
<tr>
<td>2.2</td>
<td>Clock Generation and Distribution Model</td>
<td>31</td>
</tr>
<tr>
<td>2.3</td>
<td>Translating Event Times to a Local Frame-of-Reference</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Example Signal Timing at a Positive Level-Sensitive Latch</td>
<td>34</td>
</tr>
<tr>
<td>2.5</td>
<td>Sample 4-Phase Clock System</td>
<td>35</td>
</tr>
<tr>
<td>2.6</td>
<td>Combinational Propagation Equations</td>
<td>36</td>
</tr>
<tr>
<td>2.7</td>
<td>General Synchronizer Arrival Time Constraints</td>
<td>37</td>
</tr>
<tr>
<td>2.8</td>
<td>General Synchronizer Propagation Model</td>
<td>39</td>
</tr>
<tr>
<td>2.9</td>
<td>Time Shift Function Illustration</td>
<td>41</td>
</tr>
<tr>
<td>2.10</td>
<td>Graph Model of Timing Constraints</td>
<td>43</td>
</tr>
<tr>
<td>2.11</td>
<td>Two-Bit Counter Circuit</td>
<td>44</td>
</tr>
<tr>
<td>2.12</td>
<td>Expanded Constraint Graph for Counter Circuit</td>
<td>45</td>
</tr>
<tr>
<td>2.13</td>
<td>Example Timings of Synchronizing and Non-Synchronizing Latches</td>
<td>48</td>
</tr>
<tr>
<td>2.14</td>
<td>Wave Pipelining Example</td>
<td>50</td>
</tr>
<tr>
<td>2.15</td>
<td>Wave Pipelining Due to Clock Skew</td>
<td>53</td>
</tr>
<tr>
<td>2.16</td>
<td>Wave Pipelining in Latched Circuits</td>
<td>54</td>
</tr>
<tr>
<td>2.17</td>
<td>Timing Model for Interacting Machines</td>
<td>56</td>
</tr>
<tr>
<td>2.18</td>
<td>Port Timing Behavior</td>
<td>57</td>
</tr>
<tr>
<td>2.19</td>
<td>Use of Ports to Express Input and Output Constraints</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>Algorithm SIMPLE-RELAX: Relaxation Verification Algorithm</td>
<td>61</td>
</tr>
<tr>
<td>3.2</td>
<td>Effects of Violated Loops on Relaxation</td>
<td>63</td>
</tr>
<tr>
<td>3.3</td>
<td>Modified Latch Model</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.11</td>
<td>Fraction of Latches in t953 which are Transparent after the Optimization</td>
<td>151</td>
</tr>
<tr>
<td>5.12</td>
<td>Input Ordering Example</td>
<td>164</td>
</tr>
<tr>
<td>5.13</td>
<td>$n$-input CMOS NAND gate</td>
<td>165</td>
</tr>
<tr>
<td>5.14</td>
<td>Algorithm SORT-INPUTS</td>
<td>167</td>
</tr>
</tbody>
</table>
1.1 Timing Analysis of Synchronous Digital Circuits

The design of large computers and other complex digital systems is an exercise in abstraction. Because of their large sizes, designs are partitioned into a hierarchy of levels, where each level represents an increasingly detailed view of the system. One such hierarchy is shown in Figure 1.1. The design is viewed at seven distinct levels: these are the behavioral, architectural, register transfer, gate, transistor, layout, and process levels.

Apart from the initial specification, the design process is essentially a sequence of transformations that convert representations at particular levels into equivalent representations at neigh-
boring levels. The design process has been described in terms of the direction of these transformations. When they proceed from the highest level to the lowest, the design process is referred to as top-down; a design constructed in the opposite direction is called a bottom-up design. In reality, no design fits exactly into either of these designations. Instead, each level constrains the final implementation, and representations at each level are developed and modified concurrently until the entire design is complete.

The number and complexity of constraints at each level are the key motivations for the design hierarchy. Each level relies on assumptions about the other levels of the design. These assumptions must be satisfied to guarantee correctness of the final design.

One of the most significant sets of constraints describes the timing of signals in the circuit. In high-performance systems, speed is often the single most important factor in the value and competitiveness of a circuit design; high speeds place stringent constraints on the time available for computations to take place. These timing constraints are felt throughout the hierarchy, as they both drive and limit the range of circuit performance.

Although the constraints at each level can vary in form, in virtually all cases they are used to simplify the design process and hide the complexity of the physical behavior of the circuit implementation. They reflect particular design styles and notions of “correct operation” that simplify the design process. In the higher levels of the design hierarchy, timing information is abstracted away and only the functional behavior of the circuit is designed. Progressing down the hierarchy, we add increasing amounts of physical\(^1\) detail, leading to the eventual final implementation of the circuit. Timing details are also usually abstracted away from these lower stages, so that both functional and physical properties are most commonly analyzed independently of timing constraints.

However, timing constraints have important effects on both the functional and physical aspects of a circuit’s implementation. Modules are designed with the assumption that timing constraints will be satisfied by their eventual physical implementation; an example is that a designer

\(^1\) An article by Gajski and Kuhn [47] considered the design process as operating on functional, structural, and geometrical representations. Our definition of physical detail includes all aspects of the ultimate physical implementation and does not distinguish between structural and geometrical information.
building a sequential finite state machine assumes that the machine’s next state function will be computed in time to successfully change to the next state. If a target cycle time is specified, as it usually is, the designer must rely on assumptions about the delays in the final circuit implementation. Lower-level designs (transistor netlists, chip layouts, process steps) must ensure that the final propagation delays and clock skews have values which satisfy the assumptions on which the higher-level designs (system architecture, register transfer-level descriptions) were based. This includes both intrinsic part delays and delays introduced by layout and routing effects.

One way to verify that timing constraints are satisfied is to do so by simulation; simulations can be performed at any of the levels in Figure 1.1 to determine correct timing behavior. However, simulations at higher levels of design usually rely on simplifying assumptions about lower-level details, since these details are either unknown at the time of simulation or are too complex to be handled by the simulator. Although these assumptions limit the accuracy of the result, they can still provide useful information.

Also, simulation is primarily used to verify that a structural implementation correctly matches a functional description. Little or no timing information is included. When simulation is used to verify both function and timing, timing data is added to the simulator, and the resulting signal waveforms can be checked to verify the timing behavior of the circuit.

Timing-accurate simulation is more costly than pure functional simulation and may involve redundant work if the functionality of the circuit has already been verified. A more serious problem is that to correctly verify timing properties, we must ensure that all paths are exercised in the simulation. Without additional means of analysis, we have no way of determining which paths are relevant, leaving us with the time-consuming (and often intractable) task of exercising them all.

A more effective form of verification is to perform a static timing analysis; that is, to analyze timing independently of the functional operation of the circuit. Here we assume that the functional behavior is known to be correct subject to certain constraints and assumptions about circuit timing. A static timing analysis does not consider the specific values of data signals, only whether they are stable or changing. The tasks of a static timing analyzer might be to determine whether timing constraints are satisfied (timing verification), to identify other conditions under which the
constraints are satisfied (optimal clocking), or to determine how changes in the circuit implementation will affect the timing constraints (timing-driven design).

This thesis is concerned with the static timing analysis of synchronous digital systems that use level-sensitive latches (to be defined shortly). Our work is centered on the gate and register transfer levels, although it will have implications throughout the design hierarchy. A circuit is called synchronous if all feedback paths in the circuit are broken by synchronizing devices controlled by a periodic signal called a clock. The purpose and use of synchronizing devices are addressed in detail in Chapter II, but it is sufficient here to note that these devices regulate the timing of circuit events by delaying signals until an appropriate edge occurs on the clock signal.

Our focus is on level-sensitive latches because of the complex timing properties of these devices compared to edge-triggered devices. Both types of devices are illustrated in Figure 1.2. A level-sensitive latch (or simply latch) is a synchronizing device which is sensitive to the level, or value, of a clock signal. It has three terminals: a data input, a clock input, and a data output. When its clock input is high, a positive level-sensitive latch passes signals directly from its input to its output. In this mode of operation, the latch is called transparent. When its clock input is low, the latch will hold its output value, regardless of changes on the data input. A negative level-sensitive latch is transparent when the clock is low. An edge-triggered flip-flop (or flip-flop) also has three terminals, $D$, $Q$, and $\Phi$, but its output only changes in response to a transition, or edge, in the clock signal. The output of a positive edge-triggered flip-flop becomes equal to its input when the clock signal rises and is constant at all other times; negative edge-triggered flip-flops change on the falling edge of the clock. Both types of devices require that signals on their $D$ inputs be held stable for a minimum amount of time before and after the clock switches to close the synchronizing device. These minimum times are called the setup and hold times, respectively. If these timing constraints are violated, data may be corrupted and the circuit will operate incorrectly.

Accurate timing analysis is more complicated for circuits with latches, since the times at which the device outputs change are no longer wholly determined by the clock. Figure 1.3 shows a simple circuit clocked with latches. The latches are labelled $l_1$, $l_2$, and $l_3$ and the circles represent sections of combinational logic between the latches. The maximum delays of these sections are $\Delta_{12} = 9$ and $\Delta_{23} = 6^2$. Treating the latches as if they were edge-triggered devices implies that
signals begin propagating from latch outputs on the rising edge of the clock and that they must arrive at the next latch by the next rising edge of that latch’s clock. This restrictive assumption produces a minimum cycle time of 15 as shown in Figure 1.3-a. If we require the clock schedule to be symmetric, the minimum cycle time grows to 18 (Figure 1.3-b). Note that we have assumed that the setup times are all zero.

If we instead consider the full behavior of the latch, the cycle time of a symmetric clock schedule can be reduced to 10, as shown in Figure 1.3-c. Signals begin propagating from latch \( l_1 \) on the rising edge of \( \Phi_1 \). A stable value arrives at \( l_2 \) while the latch is open and propagates directly through the latch. Finally, the stable signal arrives at latch \( l_3 \) just as the latch is closing. The cycle time is reduced because signals can arrive at each latch at any time in the interval during which its clock is high. Since signals depart as soon as they arrive at an open latch, differences in delays between latches can be absorbed to allow faster cycle times with simpler clock schedules.

Furthermore, circuit designers have more flexibility to repair possible timing violations in circuits with level-sensitive latches. For example, if the setup time of \( l_3 \) were increased to 1, there would be a 1 unit timing violation at the input of \( l_3 \) in Figure 1.3-c. Examining the timing diagram, we see that this violation could be corrected by reducing either or both delays \( \Delta_{12} \) and \( \Delta_{23} \).

---

2. The exact unit of time is not important. Typical units for delays are nanoseconds \((10^{-9} \text{ seconds})\) or picoseconds \((10^{-12} \text{ seconds})\).
This section briefly reviews existing approaches to several problems in the timing analysis of synchronous digital circuits. Three main areas are considered: timing verification, optimal clocking, and timing-driven design.

1.2.1 Timing Verification

The timing verification problem seeks to determine whether a specified circuit structure will work correctly given a desired clock schedule or timing performance. Early timing analyzers attempted to enumerate all possible paths through a circuit [1, 103, 108, 117, 141]; the timing of each path was individually checked to ensure correct operation. However, as circuits grew in size, this quickly became impractical.

1.2.1.1 Critical Path Methods

As the number of paths in circuits grew, timing analyzers began to borrow project management techniques from the field of industrial and operations engineering. In the 1950’s two sim-
ilar techniques were developed as tools for the planning of complicated design, construction, and maintenance projects: PERT (the Program Evaluation and Review Technique) and CPM (the Critical Path Method) [3, 90, 140]. Projects are described with acyclic networks called arrow diagrams, where each arrow represents a job or task that must be performed to successfully complete the project. Each arrow is labeled with the expected time needed to complete the corresponding task. In CPM, these times are represented with single worst-case values; in PERT, job times are described by a probability distribution. Vertices, or nodes, in the diagrams represent states in the project and, when joined with arrows, model the dependencies among project tasks. Two special nodes, called the source and sink, are commonly used to represent the beginning and end of a project. Without loss of generality, the source and sink are the only nodes in the network with no incoming and no outgoing arrows, respectively. The minimum project completion time is determined by the path through the project network from source to sink that requires the longest total time.

A sample project network is shown in Figure 1.4. The network corresponds to a set of tasks that might be performed at a 10-minute oil change shop. The time, in seconds, associated with each task is marked on the corresponding arrow. In Figure 1.4, node A is the source; node P is the sink. The two arrows marked with zero times (C → D and I → M) are dummy tasks [90, p. 17] that are necessary to avoid creating artificial dependencies in the project network.³

³. For example, if arrow C → D were not present, nodes C and D would be a single node, and it would appear necessary to drain the oil before the air filter could be removed! Fortunately, such complications do not arise in the circuit networks addressed here, and will not be discussed further.
The minimum time to complete a project is determined by the path through the project network from source to sink that requires the longest total time. The longest path in the network of Figure 1.4 is:

\[ A \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow J \rightarrow K \rightarrow L \rightarrow O \rightarrow P \]

Its corresponding total time is:

\[ 30 + 180 + 60 + 30 + 150 + 30 + 60 + 30 + 15 + 15 = 600 \text{ seconds} = 10 \text{ minutes} \]

In a CPM network, the minimum project completion time can be obtained by making a single pass through the network, processing each node exactly once. Beginning with the source node and proceeding in topological order, an actual event time \( e(v) \) for each node \( v \) is calculated. This is the earliest possible completion time for the entire set of jobs with arrows terminating at node \( v \) and is defined by the following equation:

\[
e(v) = \max_{u \in P(v)} [e(u) + t_{u,v}] \quad (1.1)
\]

\( e(u) \) and \( e(v) \) are the event times for nodes \( u \) and \( v \), \( t_{u,v} \) is the time of the job \( u \rightarrow v \), and \( P(v) \) is the set of node predecessors to node \( v \). Setting the event time for the source node to zero, equation (1.1) defines a unique event time for the remaining nodes. The minimum project completion time is then simply the event time of the sink node.

A critical path in a project network is a sequence of jobs that connect the source and sink nodes and whose job times determine the project completion time. Critical paths can be identified by making another pass through the network to compute required event times, \( r(v) \). Required times are defined by:

\[
r(v) = \min_{u \in S(v)} [r(u) - t_{u,v}] \quad (1.2)
\]

The required time for the sink node is set to \( T_r \), which is defined to be the required time for project completion. Required times can be calculated in a single pass that visits nodes in reverse topological order. In (1.2), \( r(u) \) and \( r(v) \) are the required times for nodes \( u \) and \( v \) and \( S(v) \) is the set of successors of node \( v \).

Together, the actual and required event times, \( e(v) \) and \( r(v) \), define a range of times for each event. The actual event time \( e(v) \) is the earliest time at which event \( v \) can occur if no job
times are reduced. The required time $r(v)$ is the latest time that event $v$ can occur without causing the project completion time to exceed $T_r$. The difference between the required and actual event times at a node is defined as the slack of the node:

$$s(v) = r(v) - e(v)$$

Note that if $r(v) < e(v)$, the slack $s(v)$ will be negative. Negative slack indicates that the project cannot be completed by the desired time $T_r$ unless one or more of the job times are reduced [140, p. 38]. A similar quantity, float, is defined for each job in the network, and is given by:

$$f(u \rightarrow v) = r(v) - e(u) - t_{u,v}$$

where $u \rightarrow v$ denotes the job between nodes $u$ and $v$. The float represents the amount by which the time of job $u \rightarrow v$ can be increased without causing the project completion time to exceed the required project completion time $T_r$.

The above definition of float has been also referred to as total float ($f_T$). Several other types of float have been defined in the classical CPM literature; each reflects a different requirement upon the way that delay increases affect the timing of jobs in the network. Each type of float is illustrated in Figure 1.5.

$$f_T(u \rightarrow v) = r(v) - e(u) - t_{u,v} \quad f_F(u \rightarrow v) = e(v) - e(u) - t_{u,v}$$

$$f_I(u \rightarrow v) = e(v) - r(u) - t_{u,v} \quad f_R(u \rightarrow v) = r(v) - r(u) - t_{u,v}$$

Figure 1.5: Types of Float in CPM Project Networks

---

4. In some texts, the actual and required event times are referred to as early and late event times, respectively. The terms actual and required are preferred here to avoid confusion with the early and late signal times introduced in Chapter II.
The total float, $f_T$, represents the maximum amount by which a job can be delayed without forcing the project time to increase. The total float values associated with each job are not independent. If the float of a job is used to increase its duration, then the floats of other jobs may as a result be reduced. The independent float, $f_I$, of a job is the amount of float which can be used without affecting the floats of other jobs. It represents the minimum amount by which a job time can be increased without reducing the required time of the initial event or increasing the actual event time of the final event. Independent floats usually represent a small fraction of the total float in a network; in fact, they are greater than zero only when there are multiple paths of jobs connecting the same pair of events.

CPM was originally used to plan large construction and engineering projects where job times were estimates subject to ongoing revision. As a result, planners were also interested in the free float, $f_F$, of a job, the amount by which the job time could increase without affecting the floats of subsequent jobs. Job times were most often revised during the job’s execution; since prior jobs had already completed, their floats were no longer relevant; it was assumed that none of their float had been used. The defining equation for free float is shown in Figure 1.5 along with a complementary notion we introduce called reverse float; this is the maximum amount of float that can be used without affecting the floats of a job’s predecessors. Obviously, $f_I \leq f_F \leq f_T$ and $f_I \leq f_R \leq f_T$. Unless otherwise specified, subsequent discussion of “float” should always be interpreted to mean the total float $f_T$.

Critical events and critical jobs can be identified as the events and jobs having the smallest slacks and (total) floats in the network. A critical path is a sequence of critical events and jobs that connect the source and sink nodes. Note that the slacks and floats of critical events and jobs can be positive, negative, or zero, according to whether $T_r - e(F)$ is positive, negative, or zero, where $e(F)$ is the actual event time at the sink node. Also note that when the critical slacks and floats are negative, the desired project completion time is infeasible unless job times are reduced until all floats are zero or positive. Alternately, a critical path is a sequence of controlling arrows connecting the source and sink nodes, where an arrow $u \rightarrow v$ is called controlling if and only if $e(v) = e(u) + t_{u,v}$.
When large project networks are analyzed, it is often possible to find multiple parallel critical paths. When parallel critical paths exist, each path is sufficient to prevent a reduction of the event time on the sink node \( F \). In order to reduce the final event time, all parallel critical paths must be shortened. If any one is unmodified, then the unchanged path will remain critical and will hold the project completion time to its original value.

Thus far we have described the activity-on-arrow (AoA) formulation for project networks. Although this is the most common formulation, another, called the activity-on-node (AoN), is also used. In the AoN formulation [140], each node in the diagram represents a task to be performed. The time required to perform each task is marked on its corresponding node. Arrows still represent dependencies between tasks; an arrow from task A to task B indicates that task A must complete before task B can begin. Analogous definitions are made for actual and required event times, slack, and float. An AoN version of the diagram of Figure 1.4 is shown in Figure 1.6.

The application of critical path analysis to digital circuits was first described by Kirkpatrick and Clark [79], and later elaborated by Hitchcock, Smith, and Cheng [59, 60]. Both groups used the AoN formulation to analyze the timing of combinational circuits, and each node in the project network represented the delay of a section of combinational logic. Kirkpatrick’s work was based on PERT, where delays were represented with probability distributions. PERT used three estimates of the delay through each logic block: the minimum, the maximum, and the most likely. It provided methods for calculating the expected delay of each path in the circuit and confidence intervals for each calculated delay. Because the path delays were statistically bounded, they were less than corresponding delays calculated using worst-case delay assumptions, allowing for more...

**Figure 1.6:** AoN Formulation of the CPM Example
aggressive timing designs. Note, however, that the statistical approach meant that some circuits 
(those lying outside the confidence intervals) would fail to work properly, and that appropriate test 
procedures were required for the final circuits.

In contrast, Hitchcock used scalar delay values and CPM, although these could be modified to represent probability distributions. As in the general CPM, critical paths were determined by a forward pass to calculate signal event times at each gate output and a subsequent backward pass to directly compute slacks (required times were calculated implicitly). But like Kirkpatrick, Hitchcock’s approach looked only at the delays through combinational blocks between storage elements to ensure that these delays were less than some maximum value required by the clock schedule, typically a distance between two edges of the clock(s) used. After isolating the combinational logic between storage elements and primary inputs and outputs, the TA program calculated the times at which signals moving through a logic block arrived at its outputs. This required that the arrival times at the inputs of the block be known, and in the TA program, they were generally assumed to be zero. This provided adequate verification for edge-triggered circuits and somewhat conservative verifications for level-sensitive circuits. Once the output arrival times had been obtained, slacks were calculated which indicated the amount of time by which signals arrived before (or after) they were required to ensure correct operation.

Although Hitchcock and Kirkpatrick both used the AoN formulation, circuit timing can also be modeled using an AoA network. In this formulation, each node corresponds to a net and arrows correspond to paths through gates. Each gate is replaced with multiple arrows instead of a single node, which provides the added flexibility to model multiple input-output delays through combinational logic blocks. Both formulations are illustrated in Figure 1.7.

We can also use CPM techniques to find the shortest path through a circuit, as described by Hitchcock et al. [59]. Now we are interested in the path from input to output with minimum delay, as this will identify the earliest time at which the output signals can change. In an AoA network, actual and required event times are calculated from:

\[
e(v) = \min_{u \in P(v)} \left[ e(u) + t_{u,v} \right] \quad (1.5)
\]

\[
r(v) = \max_{u \in S(v)} \left[ r(u) - t_{u,v} \right] \quad (1.6)
\]
Since our concern with short paths is that they not be too short (and cause hold time violations), the required time \( T_r \) becomes the earliest time that the project is allowed to complete. The definition of event times in equation (1.5) can be viewed as a replacement of the nodes in the original project network, which computed \( \text{max} \) functions, with new nodes that propagate the minimum event times from their inputs to their outputs.

To maintain the notion that negative slacks and floats correspond to errors, we simply exchange terms in the subtraction, so that the slacks and floats are defined by:

\[
s(v) = e(v) - r(v)
\]

\[
f(u \rightarrow v) = t_{u,v} + e(u) - r(v)
\]

The float on arrow \( u \rightarrow v \) represents the amount by which its delay can be reduced without causing the event time at the sink \( e(F) \) to fall below \( T_r \). Critical nodes and arrows are again defined as those having the most negative slack and float, respectively. A critical path is again a sequence of critical nodes and arrows that connect the source and sink nodes.

Designers usually check short paths when they are concerned that these paths could cause the hold time constraints on synchronizer inputs to be violated. The network is partitioned as before, and equations (1.5)-(1.8) are used to identify critical short paths. In edge-triggered circuits, problems with critical short paths are rare, and usually occur only when flip-flops have large hold
times and when zero minimum delays are assumed. If problems arise, they are typically corrected by adding delay to the short paths or using devices with smaller hold times. Hold constraints are a bit harder to satisfy in level-sensitive circuits; these difficulties are discussed in subsequent chapters.

Critical path methods are typically pattern-independent, that is, specific signal values are not considered in the analysis. Instead, only the times when signals change or become stable are computed. However, signal values can optionally be included to simplify the analysis or to possibly produce less pessimistic results. McWilliams [99] described such a method, based on a logic simulator which used a 7-valued algebra to optionally account for signal values. His SCALD timing verifier used an event-driven simulation mechanism instead of a critical path method; the approaches are similar in that they both compute signal event times by following a topological ordering of the circuit. However, SCALD’s event-driven approach allowed it to iterate over cyclic constraints, which gave it the ability to more accurately model level-sensitive latches. SCALD was also able to optionally include functional information in signal time calculations. This information allowed for more accurate verification of circuits with non-occurring input patterns or with logically impossible, or false, paths (Section 1.2.4). A summary of path-enumerative methods, a number of critical path methods (including PERT), and the SCALD verifier can be found in [7], which also describes an industrial implementation of a CPM-based verifier.

An early criticism of CPM-based techniques for circuit timing analysis was that they could only find the single worst path in a circuit [7]. However, it was later shown that one can easily enumerate all paths with delays greater than a given threshold [144]; this involves enumerating paths among parts with slacks within a threshold of the minimum slack value. Similarly, the $n$ longest paths can also be easily found [145]; a simple method for doing so is to first sort slack values in the network and then enumerate paths until the desired number of paths is obtained.

1.2.1.2 Level-Sensitive Latches

The simplest approach to the timing verification of circuits with level-sensitive latches is to treat all latches as edge-triggered devices. Signals are assumed to depart from the latch at a particular time in the clock cycle; incoming signals must not arrive later than this time. Typically, this
time is chosen to be the time when the latch is initially opened. However, it could instead be any
time in the interval in which the latch is transparent. Fixing signal departure times decouples the
timing constraints across latches, allowing the use of traditional CPM techniques.

This was the approach taken by early timing verifiers and was used in the clocking con-
straints developed by Unger [138] and Fawcett [42, 80]. Both developed sets of constraints
describing synchronous system timing and which could be used to ensure correct operation. Fawc-
ett’s work was for edge-triggered devices and a specific type of level-sensitive latch called the
Earle latch [41]; Unger’s constraints also modeled signals in both edge-triggered and level-sensi-
tive circuits. However, by fixing latch departure times, both constraint sets made pessimistic
assumptions about circuit timing which limited the accuracy of their results.

The first timing verifier to exploit the transparent property of level-sensitive latches was
described by Agrawal [1] as a part of the MOTIS timing simulator. MOTIS traced forward along
clock inputs to find latch structures embedded in a transistor-level netlist. Delays were calculated
for paths connecting latches controlled by the same clock phase and were checked to ensure that
no delays exceeded the cycle time. These paths extended through latches controlled by different
clock phases, allowing these other departure times to vary with the path delays. If a signal was
determined to arrive while an intermediate latch was enabled, the path delay calculation would
assume that the signal flowed directly through the latch to continue propagating through the next
block of combinational logic. While this successfully accounted for the transparent behavior of the
intermediate latches, it still fixed the departure times at the beginning and end of each path and
thus performed a pessimistic analysis.

The TILOS transistor-sizing program [40] also allowed signals to propagate through a
limited number of transparent latches. Signals were assumed to begin propagation when the initial
latch was enabled and were required to arrive by the enabling edge of the latch at the end of the
path. Paths were allowed to contain a user-specified maximum number of transparent latches
between the starting and final latch; this maximum was usually kept small, probably to avoid prob-
lems due to cycles of transparent latches (see Chapter III).

Jouppi’s TV (Timing Verification) program [71] used a slightly different approach which
allowed an arbitrary number of transparent latches. TV performed a number of sophisticated tim-
ing analyses, including false-path elimination, constraint propagation, delay calculation, and the enforcement of special MOS-specific constraints related to the charging and discharging of dynamic structures. However, the aspect of TV relevant to level-sensitive latches was its “borrowing” of time from one path to satisfy constraints on another. If a latch was used to separate two blocks of logic, signals from the first block could begin propagating through the second as early as the rising edge of the clock. Initially, TV assumed that all signals departed from positive level-sensitive latches on the rising edge of the clock, and that they would arrive at the next latch by the rising edge of its clock. Since signals could still be correctly latched if they arrived as late as the next falling clock edge, this was an overly-constraining assumption which could result in false indications of setup time violations. However, TV used slack values to determine when latches could “borrow” time to allow later signal arrival times. If enough slack was present, all of these artificial timing violations could be eliminated.

Another approach to the latch timing verification problem was described by Champernowne et al. [16], who presented a set of rules for determining that a latch-controlled circuit would operate correctly. These rules consider cases where latches hold up signals and where they allow them to flow through directly. However, for simplicity, they limited the number of transparent latches in the critical paths that they considered. Their rules require that no more than one flow-through latch be present in each critical path, a restriction that leads to overly-conservative diagnoses and that may result in unnecessarily slow designs.

Wallace and Sequin [139] described a similar procedure, which though it appeared earlier in time, seems to be a logical extension of Champernowne’s rule-based verification scheme. In the Abstract Timing Verifier (ATV), the clock schedule is unrolled to check paths containing up to some specified maximum number of latches. Latches internal to the path can be transparent latches, allowing paths containing an arbitrary number of transparent latches to be checked. However, unrolling loops in the clock schedule seems unnecessarily costly, and it was not clear how much unrolling needed to be done to ensure that verifications are not overly pessimistic.

Ishii and Leiserson [65] describe a method which is also based on artificially unrolling loops in a circuit (called a computational expansion) to obtain a set of conditions that guarantee correct circuit operation. Although the set of constraints they describe is potentially unbounded in
size, Ishii and Leiserson show that it can be reduced to a finite set of linear constraints which, if satisfied, will guarantee correct operation. Their formulation was among the first to allow exact verification of paths containing an arbitrary number of transparent latches. As a result, they also discovered that it was necessary to consider the possibility of loops of transparent latches, which is discussed in detail in Chapter III.

A similar approach from IBM [86] works backward from possible violations, attempting to “cycle steal” time from satisfied latches to eliminate false timing violations. The algorithm begins by labeling paths which are obviously satisfied and then backtracking through a constraint network to obtain slack to satisfy possible violations in a procedure similar to that used by Jouppi’s TV tool [71]. The backtracking algorithm also checks for loops of transparent latches, but the authors do not describe the computational cost of this potentially expensive operation.

Most of the previous work on timing verification was focused on ensuring that the delays in a circuit are small enough to guarantee correct operation at some desired cycle time. However, when latch-controlled circuits are used, it is also important to verify that delays are large enough to prevent hold violations at latch inputs. One way to verify this is to use a clocking methodology that eliminates hold violations a priori. The most common such methodology is to control a circuit with a pair of non-overlapping clocks and assign clock phases so that no two consecutive latches are controlled by the same clock phase [102]. This ensures that no two consecutive latches can be open at the same time. If clock skew and hold times are zero, this eliminates the possibility of hold time errors. Nonzero skew and/or nonzero hold times can place minimum separation constraints between clock phases that ultimately reduce circuit speed and complicate the design process.

The verification methods we consider are based on those developed by Sakallah, Mudge, and Olukotun in the checkTc program [116]. They presented an algebraic model for signal timing in circuits with level-sensitive latches and proposed an iterative technique for solving these equations to determine satisfaction of the timing constraints. Both the latest and the earliest signal times were computed, allowing verification of both setup and hold constraints. In related work, Szyman- ski and Shenoy [133] showed that certain pathological cases could cause the checkTc algorithm to converge very slowly; they then described modifications to improve performance and showed that the timing constraints could be solved using a variant of the Bellman-Ford algorithm [27] for solu-
tion of sets of linear inequalities. An extended version of the “SMO” model is presented in Chapter II; its application to timing verification is discussed in Chapter III.

1.2.2 Optimal Clocking

The problem of *optimally clocking* a circuit is similar to the timing verification problem, but instead of testing a predetermined clock schedule, optimal clocking algorithms find the best clock schedule subject to a given set of constraints. This can involve determination of the cycle time and the relative position of clock events (rise and fall times) within the clock schedule.

For edge-triggered circuits, we can solve optimal clocking problems using the CPM techniques described Section 1.2.1.1. The goal in this case is to determine, for the given circuit, the clock schedule with the smallest possible cycle time, $T_c^{\text{min}}$. Thus, the required time for each section of combinational logic is set to the maximum path delay through the section. Adjusting for setup times, these required times determine the spacing of events in the optimum clock schedule, including the minimum clock period.

The Crystal timing analyzer [107] performed optimal clocking for nMOS circuits using long-path delay information which it extracted from transistor layouts. Neglecting synchronizer setup times, Crystal calculated the minimum separation between clock phases based on the delays between synchronizers controlled by these clock phases. Crystal used a number of sophisticated techniques for delay calculation and false path elimination, but used a very simple clocking model. As in our work, specific signal values were ignored. Devices were assumed to be edge-triggered, and minimum delays between synchronizers were ignored.

The problem of optimally clocking latch-controlled circuits was first considered by Agrawal [1], who used a timing verifier in a binary search to find the minimum cycle time. A more analytic approach was taken by Szymanski in the LEADOUT program [131], which generated constraints between events in the clock schedule that specified the separations that would guarantee correct circuit operation. These constraints could be checked against the calculated circuit performance to verify circuit timing and, if clock duty cycles were fixed, could be solved symbolically to obtain the minimum error-free cycle time.
For more complex optimizations which allowed clock duty cycles to vary, the Pearl program [24] generated a set of constraints between events similar to LEADOUT’s but used a linear programming algorithm to solve the optimization problem. However, although examples included circuits with level-sensitive latches, a timing model for latch behavior was not included, suggesting that latches were treated as edge-triggered devices in the analysis. This also seems to be reflected in Pearl’s assumption that clock edges are the primary causes of signal transition events.

Another linear programming approach for edge-triggered circuits was described by Fishburn [44]. Beginning with an edge-triggered circuit controlled by a single-phase clock, Fishburn applied a variable clock skew to each flip-flop and used a linear program to find the clock skews which would minimize the cycle time. This method produced minimum cycle times, but required difficult-to-obtain control over the clock distribution network and introduced implicit wave-pipelining effects (see Section 2.4.3.1).

Dagenais and Rumin [31] observed that the optimal clocking problem for level-sensitive latches could also be expressed as a linear program, with the cycle time as the quantity to be minimized. However, because of the presumed cost of such LP solutions they instead developed a relaxation-based algorithm which iteratively approached the optimal solution. They defined a quantity called the retardation at each latch in the circuit, which is the difference between the actual time signals stabilize on the output of a latch and the earliest possible time, which is specified by the event in the clock schedule that opens the latch. Their algorithm begins by assuming that there is no retardation at each latch in a circuit, or in other words, that all signals arrive before the latch is enabled by the clock. This decouples the paths between latches, greatly simplifying the analysis process. The proposed algorithm then attempts to increase the retardation at latches on the critical path to allow the cycle time to be reduced. However, the amount of increase in retardation values is not specified, and the algorithm as implemented only performs a single iteration, making it impossible to guarantee that the true optimum will be found.

Sakallah, Mudge, and Olukotun carried this approach a step further by approaching the optimal clocking problem as a mathematical optimization problem that could be solved by general-purpose techniques. Using the same models as they used for the verification problem, they formulated linear programs which could be solved to give the clock schedule which corresponds to
the minimum cycle time [114, 116]. Unlike previous work, their models also presented an accurate picture of short path effects and hold time constraints. But since these constraints involved min and max functions, linear programs could only be obtained by relaxing the min and max functions, or replacing them with corresponding inequalities. As discussed in Chapter IV, this relaxation underconstrains the original problem, and in some cases makes it impossible to find solutions using linear programming. This was handled by restricting the original constraints to a smaller feasible region, which can be shown to be exactly solvable with an LP. A second problem was the large number of constraints, making the method impractical for large circuits. Jennings et. al. used a fast interior point LP solution algorithm to increase the range of feasible circuit sizes [69], but the largest improvement in performance was due to a reduction developed by Szymanski. He used a graph representation of the timing constraints to obtain a dominant set of constraints between clock events which resulted in significantly smaller LPs [132]. Shenoy then added the observation that these smaller LPs could be solved using the Bellman-Ford algorithm in a binary search over the range of possible cycle times [120].

1.2.3 Timing-Driven Design

When a circuit fails to satisfy its timing requirements, designers are forced to consider changes to the design that will allow it to operate correctly. Similarly, designers often begin with an initial timing-feasible design and seek to transform it into a smaller or lower power circuit that still satisfies its timing constraints. The timing-driven design problem covers both situations, and seeks to improve some aspect of a circuit design while maintaining or improving its timing performance.

A wide variety of approaches have been developed for timing-driven design, and can be characterized by the level of design at which they are applied. Because of the scope of our work, we focus on optimization techniques used on designs ranging from the register transfer to the layout level (Figure 1.1). At the highest level, there are a number of techniques based on changing the structure of the circuit. At lower levels, there are optimizations based on changing the devices used to implement each specified function. Finally, we consider optimizations based on changes in the placement of parts and the routing of the lines that connect them.
Virtually all of these approaches require some notion of criticality to guide the optimization, and in most cases, the Critical Path Method is again applied to determine which parts of a circuit satisfy their timing constraints and which do not. More importantly, slack and float values identify the amount by which delays must be modified to repair timing violations or the allowable changes that will not cause constraint violations.

1.2.3.1 Logic Resynthesis

Logic synthesis and resynthesis optimizations rely on substitutions that change the structure of the gates in a circuit while preserving the behavior seen at the circuit outputs. Because of the complexity of this problem, most logic synthesis techniques are based on heuristic methods, although a few treat the problem as an optimization which can be solved with linear programming or branch-and-bound techniques [32, 73]. Most other approaches rely on iterative application of algebraic transformations that factor and expand boolean expressions to alter the number of logic levels in a circuit [5, 12, 23, 33, 126]. Transformations are usually focused on a subsection of a circuit called an \( \varepsilon \)-network, which is the subset of parts with slack values within \( \varepsilon \) of the smallest slack in the circuit. Related areas that use slack information and \( \varepsilon \)-networks include the mapping of logic functions to gates (technology mapping) [76], the partitioning of gate inputs based on signal arrival times [61], and the addition of buffers to help drive large loads [127].

Recently, Shenoy presented a framework for the application of algebraic logic optimizations to pipelined circuits clocked with level-sensitive latches [122]. After defining slacks on each gate in a circuit, he uses a quadratic program to minimize the Euclidean distance between the original design point (specified in terms of the delay values) and a new point which satisfies the timing constraints. This new design point specifies target delays for algebraic speedup operations. This approach assumes that the best set of target delays is the one nearest the original delays; although no proof of this assumption is presented, it seems intuitively justifiable. However, the cost of the quadratic program solution apparently limits this approach to moderately sized circuits with a small number of pipeline stages, suggesting that less expensive heuristics may be available that produce comparable results.
1.2.3.2 Retiming

Retiming optimizations preserve the structure of the logic gates in a circuit but allow storage elements to be moved across logic to minimize the cycle time or the number of storage elements. In [84], Leiserson described the original retiming model which he developed for optimizing single-phase, edge-triggered circuits and presented a set of efficient, well-characterized algorithms for finding optimal retimings with respect to both cycle time and register count. One such algorithm can be viewed as an iterative procedure that repeatedly shortens the most constraining paths in the circuit and can be used to quickly obtain a feasible retiming, if one exists. However, a number of restrictions were made to make the algorithms tractable. For example, the gate-level structure is fixed throughout the optimization, causing retiming algorithms to miss optimizations which can be performed by making changes to circuit logic structure. In addition, the restriction to single-phase clocking and edge-triggered flip-flops renders the algorithms unsuitable for fully optimizing multiphase circuits or circuits using level-sensitive latches.

In [34], De Micheli addressed the first restriction in Leiserson’s work and identified simple logic transformations that correspond to various retiming operations. Although they were essentially guided by heuristics, these operations were well defined and again used criticality information to determine parts of the circuit to be modified. Malik and Sentovich [92] also described an approach which can allow for some logic reorganization within and between logic blocks. They define a peripheral retiming of a circuit as one which moves all of the registers in the circuit to the circuit perimeter. The remaining logic is then reoptimized using a standard tool such as misII [12] and the peripherally retimed registers are then moved back into the circuit. Another heuristic extension was described by Bartlett and Borriello [6] for multiphase circuits clocked with edge-triggered flip-flops. They used estimated minimum delays through logic blocks as a heuristic to guide the movement of logic across register boundaries. Iqbal et. al. also combined Leiserson’s classical retiming algorithm with algebraic speed-up techniques [64]. This approach used an algorithm developed by Dey [36] to modify circuit structure to improve retiming results.

Retiming under more complex clocking schemes was considered by several other researchers. Shenoy [118] proposed a model for one-phase retiming of level-sensitive latches, but the complexity of the single-phase latch constraints made his approach impractical. In separate work, Lock-
year [89] and Ishii [66] developed exact methods for multiphase retiming of circuits with level-sensitive latches. Both methods placed fairly strenuous constraints on the types of circuits to be optimized, and both again used notions of critical paths to guide their optimizations.

1.2.3.3 Transistor Sizing

In MOS technologies, signal delays are directly related to the amount of current which can be driven by a part, which in turn is determined by the sizes of transistors that make up the part. However, as transistor sizes increase, circuit area and power dissipation also increase, creating a tension between timing optimization and area and power minimization. A further complication arises when we observe that as transistor sizes increase, their corresponding gate capacitances also increase, increasing the delay of parts fanning into the larger transistor. This means that the fastest circuit may not be the one composed of the largest transistors, as instead a balance must be kept between drive capability and input loading.

For simple circuit structures, exact solutions that provide maximum delay for a given area constraint can be found [52, 55, 56]. However, in general, the transistor sizing problem is a difficult non-convex optimization problem which has in practice been solved by heuristic algorithms [74, 136, 143], simulated annealing [19], or mathematical programming approaches based on linear programming approximations [10], quadratic programming [43, 124], Lagrange multipliers [26, 95], or other nonlinear optimization techniques [22, 96]. Each of these approaches provides a different quality of result that is generally inversely related to computational cost, with the most “exact” algorithms often being impractical for circuits of more than a few hundred transistors. The problem is greatly simplified when changes in input capacitances can be ignored [33, 111]. In all cases, information about part criticality is essential, and is usually provided within a CPM-style framework. In fact, many of the above procedures work on a single path at a time, where each optimization step alternates with a CPM determination of the next critical path to optimize.

1.2.3.4 Part Selection

The part selection problem is closely related to that of transistor sizing. However, instead of optimally sizing individual transistors it involves the selection of parts from a catalog. Typi-
cally, parts are taken from commercial standard cell libraries [135] that contain a variety of implementations, or variants, of each gate type, where each part variant has a different area, drive capability, input load, and power dissipation. A typical optimization problem is to select one of a discrete set of variants for each part to minimize area subject to a time constraint or to optimize timing subject to a maximum area constraint. Because of the interaction of drive capability and input capacitance, the general part selection problem has been shown to be NP-hard [17, 57]. As a result, most part selection algorithms are based on heuristics [104], some of which use sensitivity information to guide the selection of parts for substitution [87]. A key factor in these sensitivity equations is the slack associated with each part, obtained using CPM. Other methods repeatedly apply exact algorithms for optimal sizing of part chains [58] or fanout-free trees [17, 57], where the subnetwork to optimize is identified again using critical path techniques. A more global view is taken in a method developed by Chuang et. al. [25] which approximates the optimization as a linear program and then maps the solution back to parts in the library; however, their algorithm still relies on CPM techniques to guide the mapping and final timing verification.

1.2.3.5 Input Ordering

Another timing optimization is based on the observation that the structure of CMOS gates causes paths from different inputs to the gate output to have different lengths. As a result, each gate input can have a distinct delay to the output node. Although most analysis has indicated that the latest arriving signals should be placed nearest to the output node [21, 40, 94], more recent studies suggest cases where other orderings are better [15]. However, in all cases, ordering is done by computing arrival and required times in a CPM network and assigning the inputs with the smallest delays to the paths with the least slack. So as with the other optimization techniques, critical path methods again provide the basic framework for these optimization approaches.

1.2.3.6 Placement and Routing

The lowest level of design we consider is the placement of parts that make up a design and the routing of the nets, or wires, that connect them. If the parts to be used and their interconnec-
tions are fixed, the placement of parts and the routing of their interconnect can still have significant effects on circuit timing.

A number of different approaches have been taken to the problem of incorporating timing constraints into part placement and routing. In the simplest, nets are weighted so that preference is given to reducing the length of nets that have smaller slack values [14, 39, 93, 134] or that lie on paths with timing violations [48]. Slacks and possible violations are calculated using CPM techniques. Experiments and practice show good results for each of these methods, but in general, they do not guarantee that timing constraints will be satisfied by the final physical design, as lower-priority paths can be made critical in the placement and routing and can cause new violations.

A more reliable approach is to use slack information to place bounds on net lengths, such that if a place and route tool successfully satisfies the net length bounds, the timing constraints will be guaranteed to be satisfied. An important concern in these approaches is the allocation of slack among the nets on a path, as the slack of a path must be divided among multiple nets. A number of algorithms exist, including the zero-slack algorithm [54, 91, 101, 137], the iterative minmax-PERT algorithm [146, 147], and the limit-bumping algorithm [45]. Each of these algorithms use weighting functions to guide the slack allocation, where weights are based on fanout counts, load capacitance, device drive capability, and other physical design characteristics. In other cases, scoring functions are computed to identify and prioritize potentially problematic paths [148].

Other approaches use incremental analysis to evaluate moves to perform global [38] or hierarchical [67, 130] placement. Placement can also be treated as a mathematical programming problem; [68] and [129] describe nonlinear programming approaches which use a nonlinear cost function for net lengths. In each case, the mathematical programs include constraints which are equivalent to the CPM event and required time equations (1.1) and (1.2).

1.2.4 The False Path Problem

A significant problem in timing analysis which we do not address here is the false path problem [8, 20, 35, 63, 97, 125]. Critical path methods can occasionally identify false paths over which it is logically impossible for signals to flow. Since these paths are never activated, or sensi-
tized, the timing constraints associated with such paths are false, and should be ignored in the timing analysis.

Because of its complexity, we ignore the false path problem in this work, focusing instead on designing timing analysis procedures that can be later adapted to include techniques for eliminating false paths. One such example is described in [70], where Jone and Fang describe a resizing procedure which uses false path information. After calculating the circuit timing, Jone and Fang use the PODEM algorithm [53] to identify the subnetwork of longest sensitizable paths. Their algorithm then picks the gate whose resizing will produce the maximal reduction in path delay, resizes it, and repeats the above steps.

### 1.3 Thesis Organization

This dissertation describes the development and application of a timing model for synchronous digital circuits clocked with level-sensitive latches. This model is designed primarily for static timing analysis. As a result, it is data-independent, in that it does not depend on the actual values of signals being analyzed, and instead considers only the times at which events occur in a circuit.

After presenting the model, we demonstrate its usefulness by using it as a framework for discussing several timing issues which arise in digital circuits and as a basis for solving a number of significant timing analysis problems, including timing verification, optimal clocking, and the systematic modification of circuit delays to meet desired performance constraints.

A pictorial overview of the thesis is presented in Figure 1.8. In Chapter II, we discuss the basic timing model we use, which is an extension of the one developed by Sakallah, Mudge, and Olukotun [114]. We show how to account for minimum and maximum gate delays, clock skew, and the full range of behavior of level-sensitive latches. We also discuss the uses of clocks in a synchronous circuit and define several terms that are useful for classifying clocking methodologies and modes of latch operation.

We begin Chapter III by describing a very simple procedure for verifying the timing of a synchronous digital circuit. We then address the problem of identifying the critical paths which can arise in circuits with level-sensitive latches and describe the extension of critical path methods
to these circuits. These paths place the most stringent constraints on circuit operation and, as we shall see, in circuits with level-sensitive latches several types of critical paths can arise. We classify these critical paths and present procedures for identifying them.

In Chapter IV we address the problem of optimally clocking a digital circuit. We consider the constraint reduction of Szymanski in light of the CPM extension of Chapters II and III, and we present an analysis of the original set of SMO constraints. In our analysis, we show that the original constraints belong to a class of problems which we call min-max linear programs (mmLP). After discussing some characteristics of these problems, we present conditions that identify cases which can be solved using a single linear program. In addition, we describe a branch and bound algorithm for the original problem. We then use this branch-and-bound algorithm to find exact solutions of optimal clocking problems for arbitrary circuits of non-trivial size. This allows us, for the first time, to compare exact optimal clocking solutions with those obtained under the more restrictive constraints which were added to simplify the solution space [116].

Chapter V extends the critical path definitions of Chapter III and applies them to a pair of timing-driven design problems. We compare a number of existing CPM-based algorithms with extended versions which operate on the extended critical paths. We also present simple schemes for handling cyclic constraint networks.

Finally, in Chapter VI we summarize the contributions of this work and discuss areas for future research.
2.1 Timing Modeling

As system performance goals increase and circuit feature sizes shrink, a correct and precise understanding of timing issues becomes crucial for ensuring that large digital systems work as specified and at the necessary speeds. As a result, it is important that models and methods be developed to verify that circuits work as expected and to correct or optimize them when their timing behavior contains errors or needs to be improved.

This chapter describes a model for the static timing analysis of digital circuits. The timing behavior of large sequential systems is modeled and analyzed in a data-independent manner; it is assumed that all functional verification is handled separately, subject to a set of assumptions about circuit timing which must be satisfied. These assumptions are expressed in the timing models and definitions developed in this chapter. In subsequent chapters we apply these models and definitions to several analysis and optimization problems.

The timing model presented here is an extension of the one developed by Sakallah, Mudge, and Olukotun [105, 114, 116]. The original “SMO” model described the behavior of positive level-sensitive latches and negative edge-triggered flip-flops. Combinational delays between synchronizing devices were lumped together, and primary inputs and outputs were assumed to be connected through latches. Signal times were expressed in a local frame of reference defined by the clock controlling a corresponding synchronizer.

This chapter describes several extensions to the SMO model. We simplify the synchronizer model to more easily allow for both positive and negative synchronizing device types. We
add terms which account for delays and uncertainty in the clock distribution network. We provide a more flexible model of input and output signal timing. We introduce a graph representation which will later prove useful in the identification and optimization of critical paths. We also present an extension of the general model which allows for individual accounting of combinational delays.

A number of other modeling-related issues are also addressed here, including modifications to the model to support wave-pipelining and the definition of several terms related to circuit timing behavior which clarify and elaborate important timing properties.

### 2.2 A Timing Model for Synchronous Circuits

This section describes the model we use for synchronous circuits clocked with both level-sensitive and edge-triggered devices. Circuits are partitioned in a way similar to that shown in Figure 2.1 [51], in which circles represent combinational logic, squares labeled $s_i$ represent storage devices (also called synchronizers\(^1\)), and arrows represent connections between elements and indicate the direction of data flow in the circuit. Input and output ports ($i_i$ and $o_i$, respectively) are also shown.

The following subsections describe our model for synchronous circuits in detail. Section 2.2.1 describes the model parameters, Section 2.2.2 presents the model constraints, and a graph representation of the constraints is introduced in Section 2.2.3.

---

\(^1\) In this thesis, the terms *synchronizer* and *storage element* are used interchangeably.
2.2.1 Model Parameters

We describe sequential systems in terms of three types of variables: clocking parameters which model clock generation and distribution, structural parameters that include circuit propagation delays and setup and hold times, and signal event times which describe the times at which signals may change in a circuit.

2.2.1.1 Clocking Parameters

The clock model is illustrated in Figure 2.2. The model describes multiphase systems having a common clock cycle time, $T_c$. Each clock signal rises and falls exactly once per cycle, and the times of these events are specified by the event times $R_p$ and $F_p$, respectively, where the corresponding clock signal is denoted $\Phi_p$. These event times are with respect to a common system frame of reference that is modulo $T_c$; they must lie in the interval $(0, T_c]$. Together with the cycle time, $T_c$, a complete set of rise and fall times for the clock signals used is called a clock schedule.

The clock lines connected to each synchronizer can have delays associated with them; $q_i$ and $Q_i$ are the minimum and maximum delays from the clock generator (or the ideal clock signals) to the clock input of synchronizer $s_i$. Separate values for $q_i$ and $Q_i$ allow the modeling of uncertainty in the clock generation and distribution network.

In the following we use the terms “clock signal” and “clock phase” synonymously, that is, a clock phase is defined as a distinct periodic signal produced by the clock generation subsystem. A more accurate use of the term “phase” might be to define a clock phase as the time interval during which a corresponding clock signal is high (or low). However, for clarity and consistency with current usage, we keep our original definition of a clock phase as an independent clock signal.

2.2.1.2 Circuit Structural Parameters

The circuit parameters are derived from the structure of the circuit and include the setup and hold times for each synchronizer ($S_i$ and $H_i$, respectively), the minimum and maximum delays
through the synchronizers from their clock ($\delta_{C_i}$ and $\Delta_{C_i}$) and data ($\delta_{D_i}$ and $\Delta_{D_i}$) inputs, and the minimum and maximum delays ($\delta_{i,j}$ and $\Delta_{i,j}$) through the combinational logic connecting synchronizer $s_i$ to synchronizer $s_j$.

Synchronizers are described by their enabling ($E_i$) and latching ($L_i$) event times which, in turn, correspond to events in the clock schedule. Since they correspond to clock events, the enabling and latching events occur once per cycle with period $T_c$. The enable event causes data to move from the inputs to the outputs of the synchronizer and the latch event causes the data on the outputs to be held constant. The primed versions of these variables ($E'_i$ and $L'_i$) denote their values in a local frame-of-reference which begins and ends on the latching event time $L_i$; their values are obtained using the equation $t' = T_c - (L_i - t) \mod T_c$, which relates an event time $t$ in the global frame of reference to a time $t'$ in the frame of reference of synchronizer $s_i$. This can also be

\[^2\] Alternately, synchronizer delays may at times be represented as simply $\delta_i$ and $\Delta_i$. 

---

Figure 2.2: Clock Generation and Distribution Model
expressed as $T_c - t' = (L_i - t) \mod T_c$ and is illustrated in Figure 2.3. For the enable event, this gives $E'_i = T_c - (L_i - E_i) \mod T_c$ and for the latch event, $L'_i = T_c - (L_i - L_i) \mod T_c = T_c$.

By allowing the enabling and latching events to be associated with either the rising or falling clock edges, we can specify four different synchronizer types, as shown in Table 2.1. This allows us to use the same notation to describe both positive and negative versions of edge-triggered flip-flops and level-sensitive latches.

<table>
<thead>
<tr>
<th>Synchronizer Type</th>
<th>Symbol</th>
<th>$E_i$</th>
<th>$L_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Level-Sensitive Latch</td>
<td><img src="image" alt="Positive Level-Sensitive Latch" /></td>
<td>$R_{p_i}$</td>
<td>$F_{p_i}$</td>
</tr>
<tr>
<td>Negative Level-Sensitive Latch</td>
<td><img src="image" alt="Negative Level-Sensitive Latch" /></td>
<td>$F_{p_i}$</td>
<td>$R_{p_i}$</td>
</tr>
<tr>
<td>Positive Edge-Triggered Flip-Flop</td>
<td><img src="image" alt="Positive Edge-Triggered Flip-Flop" /></td>
<td>$R_{p_i}$</td>
<td>$R_{p_i}$</td>
</tr>
<tr>
<td>Negative Edge-Triggered Flip-Flop</td>
<td><img src="image" alt="Negative Edge-Triggered Flip-Flop" /></td>
<td>$F_{p_i}$</td>
<td>$F_{p_i}$</td>
</tr>
</tbody>
</table>

Table 2.1: Synchronizer Types
Input and output ports are also shown in Figure 2.1; their timing behavior will be described in the next section.

### 2.2.1.3 Data Signal Timing Parameters

The data model which we use describes timing properties only; as such, it does not record particular signal values, but only whether a signal is stable or changing at given times. Also, since we are primarily interested in the interaction of data with clock signals, the data variables in the basic model describe signals at the inputs and outputs of synchronizers. The variables that we use are as follows:

- \( a_i \): the earliest time that a data signal can change at the input of synchronizer \( s_i \).
- \( A_i \): the latest time that a data signal can change at the input of synchronizer \( s_i \).
- \( d_i \): the earliest time that a data signal can change at the output of synchronizer \( s_i \).
- \( D_i \): the latest time that a data signal can change at the output of synchronizer \( s_i \).

Within the intervals \([a_i, A_i]\) and \([d_i, D_i]\) signals are considered to be possibly changing; outside these intervals they are known to be stable. These *arrival* and *departure* times are relative to a frame-of-reference associated with controlling clock signal for synchronizer \( s_i \). Each frame-of-reference begins and ends on the latching edge of a clock signal. *Time shift functions* \( \mathcal{O}(e_j, e_i) \), to be defined shortly, are used to transform signals described in the frame-of-reference of defined by event time \( e_j \) to that of event time \( e_i \). A sample timing diagram is shown in Figure 2.4; signal timings are described by the bars placed below the clock signal. The white section of the bar specifies the time interval during which the signal is stable at its old value (determined in the previous cycle), the dark section of the bar corresponds to the time interval during which the signal can be considered changing or unstable, and the lightly shaded section indicates that the signal is stable at its new value.

Input port timings are specified by the earliest and latest times that data can depart from the port into the circuit being analyzed. Similarly, output ports are specified by the earliest and lat-

---

3. Unless otherwise noted, the variables \((a_i, A_i, d_i, D_i)\) refer to times in the local frame of reference of synchronizer \( s_i \). Primes are omitted to simplify the notation.
the times at which signals must arrive at the port from the circuit being analyzed. These departure and arrival times must be defined with respect to one of the clock events in the system; this event identifies the frame-of-reference for the input or output port.

### 2.2.2 Model Constraints

The timing constraints that model system timing can be classified into three groups. The first set of constraints describes limitations which can be placed on the clocks in the circuit, the second describes the propagation of data through combinational logic, and the third set models the timing behavior of the synchronizers in the circuit.

#### 2.2.2.1 Clock Constraints

Clock constraints express basic limitations on clock generation and distribution. This set should at least include constraints on the minimum width of a clock pulse that can be generated and distributed:

\[
(R_p - F_p) \mod T_c \geq w
\]  

(2.9)

\[
(F_p - R_p) \mod T_c \geq w
\]

(2.10)

where \(w\) is a specified parameter. In addition, to simplify the design of the clock generator we may include regularity constraints such as the example set listed below.
These constraints describe the four-phase system shown in Figure 2.5, which can easily be obtained by dividing a clock with period $\frac{1}{4}T_c$ or by using a phase-locked loop to multiply a lower-frequency clock [30]. Note that these extra constraints are included only to simplify the clock generator design. In general, we do not require that clock phases be non-overlapping or be of uniform width. However, in some cases constraints like these will be used to simplify analysis and optimization procedures.

2.2.2.2 Combinational Propagation Equations

These equations model the delay of the combinational sections of the circuit connecting internal synchronizers and circuit inputs and outputs. They express the arrival times of data at synchronizers $s_i$ (or output ports $o_j$) in terms of the departure times from synchronizers $s_j$ (or input ports $i_j$) that fan signals into them:

$$a_i = \min_{j \in F(i)} (d_j + \delta_{j,i} - \emptyset (L_j, L_i))$$  \hspace{1cm} (2.12)
where \( FI(i) \) describes the set of synchronizers and primary inputs fanning into \( s_i \) (or \( o_i \)). Equations (2.12) and (2.13) are illustrated in Figure 2.6.

![Figure 2.6: Combinational Propagation Equations](image)

\[
A_i = \max_{j \in FI(i)} (D_j + \Delta_{j,i} - \emptyset(L_j, L_i))
\]

(2.13)

2.2.2.3 Synchronizer Macromodels

Synchronizer macromodels are comprised of two types of relations: constraints which guarantee hazard-free operation and equations which describe the input-output behavior of the synchronizers.

**Signal Hazard Constraints** (also called Latching Constraints) express the conditions necessary for correctly capturing data values at each of the synchronizers. Typically, they consist of two requirements that, together, ensure that arriving data signals are stable for a sufficient time before and after the latching edge of a synchronizer’s clock. If \( S_i \) is the setup time and \( H_i \) the hold time of synchronizer \( s_i \), these setup and hold constraints can be expressed as:

\[
A_i = \min(d_j + \delta_{j,i} - \emptyset(L_j, L_i), d_k + \delta_{k,i} - \emptyset(L_k, L_i), d_l + \delta_{l,i} - \emptyset(L_l, L_i))
\]

(2.14)

\[
A_i = \max(D_j + \Delta_{j,i} - \emptyset(L_j, L_i), D_k + \Delta_{k,i} - \emptyset(L_k, L_i), D_l + \Delta_{l,i} - \emptyset(L_l, L_i))
\]

(2.15)

The synchronizer arrival time constraints are illustrated in Figure 2.7.

Circuits controlled by level-sensitive latches are especially prone to hold time violations. An observation of this fact was made by Shenoy [118], who looked at the problem of retiming cir-
circuits controlled by single-phase latches. In retiming algorithms, storage elements are moved back and forth across logic gates in a circuit in an attempt to reduce cycle time, area, or some other parameter of interest. Shenoy et al. observed that for single-phase latch retimings, the number of latches on a path between a pair of logic blocks must be either zero or one. If an additional latch were present, its hold time would be violated and its contents would be identical to the data stored in the first latch on the path. In other words, this second latch would be redundant and its removal would have no effect on circuit operation. Hold violations can also arise in a number of more sophisticated cases, and have been discussed at least as far back as 1965 [28]. Because these violations cause more than one wavefront of data to pass through a latch during a single clock cycle, they are also referred to as double-clocking errors [44]. A similar term, zero-clocking, is often used to describe violations of the setup time, which prevents a valid signal from being clocked into a latch.

**Synchronization Equations** express the departure times of output data signals in terms of the arrival times of corresponding input data signals and the rising and falling edges of the synchronizer’s clock.

As shown in Table 2.1, edge-triggered devices use the same clock event for both their enabling and latching events. Thus new signals can begin propagating from the latch at the time $E_{i}^{'} = L_{i}^{'} = T_{c}$. Including clock delays and the latch clock-to-output delays gives:
For level-sensitive latches, the enabling and latching events are separate events in the clock schedule. Once a latch is enabled, data can propagate freely through the latch until the latch is closed again. This relationship is expressed by the following equations:

\[
d_i = T_c + q_i + \delta_{Ci} \tag{2.16}
\]

\[
D_i = T_c + Q_i + \Delta_{Ci} \tag{2.17}
\]

These equations are illustrated in Figure 2.8. Figure 2.8-a shows the original clock signal \(\Phi_{pi}\) and the clock signal as seen by synchronizer \(s_j\). Figure 2.8-b, Figure 2.8-c, and Figure 2.8-d show signal timing when signal departure times are determined by the enabling event, the signal arrival times, and the latching event, respectively.

Observing that \(E'_i = L'_i = T_c\) for edge-triggered devices, equations (2.18) and (2.19) reduce to the flip-flop equations (2.16) and (2.17), respectively.

For most problems we assume that the latch is operating properly subject to the satisfaction of the setup and hold time constraints. In these cases, we are not interested in the behavior of the latch when signals arrive too late, other than to identify this as a timing error. Thus we can eliminate the min functions and express the departure time equations as simply:

\[
d_i = \max(a_i + \delta_{Di}, E'_i + q_i + \delta_{Ci}) \tag{2.20}
\]

\[
D_i = \max(A_i + \Delta_{Di}, E'_i + Q_i + \Delta_{Ci}) \tag{2.21}
\]
Figure 2.8: General Synchronizer Propagation Model

a. Clock Distribution Timing

b. Latch Timing (Enabling Event Limited; signals arrive before latch is enabled)

c. Latch Timing (Arrival Time Limited; signals arrive while latch is open)

d. Latch Timing (Latching Event Limited; signals arrive after latch is closed)
2.2.2.4 Time-Shift Function

As mentioned previously, an important part of our timing model is the time shift function \( \varnothing (e_j, e_i) \). This function describes the time difference between a pair of reference frames, each defined by a clock event time. \( \varnothing (e_j, e_i) \) is given by:

\[
\varnothing (e_j, e_i) = \begin{cases} 
(e_i - e_j), & \text{if } (e_i > e_j) \\
T_c + (e_i - e_j), & \text{if } (e_i \leq e_j)
\end{cases} 
\] (2.22)

Since \( 0 \leq e_i < T_c \) and \( 0 \leq e_j < T_c \), we can also represent this as:

\[
\varnothing (e_j, e_i) = T_c - (e_j - e_i) \mod T_c 
\] (2.23)

By requiring that \( 0 < \varnothing (e_j, e_i) \leq T_c \), this definition guarantees that signals departing from synchronizers with a latching event time of \( e_j \) will arrive at devices with latching event \( e_i \) before the next occurrence of \( e_j \). An illustration of the time shift function is given in Figure 2.9.

Other functions could be used to model different definitions of correct operation. For example, the timing constraints for a two-cycle path could be obtained by adding \( T_c \) to the right-hand side of equation (2.23), and the constraints for a zero-cycle path \( l_j \rightarrow l_i \) (where data is assumed to be latched into the output latch \( l_j \) at the same time it is latched into the input latch \( l_i \)) can be obtained by setting \( \varnothing (e_j, e_i) = 0 \).

2.2.3 Graph Representation of Constraints

The main components of our timing model are summarized in Table 2.2. As shown, these constraints can be separated into two independent sets: one for early signals \( (a_i \) and \( d_i) \) and one for late signals \( (A_i \) and \( D_i) \).

Each of these constraint sets can be represented by a graph, as shown in Figure 2.10. In both graphs, each latch is represented by a pair of nodes labeled \( A_i \) and \( D_i \), or \( a_i \) and \( d_i \), which correspond to the arrival and departure time equations for the latch. A \( \delta_{D_i} \)- or \( \Delta_{D_i} \)-weight arc co-
nects the arrival and departure time nodes and reflects the arrival time terms in the latch departure time equations (2.31) and (2.32). These arcs are not present for edge-triggered synchronizers, just as there are no arrival time terms in the flip-flop departure time equations (2.33) and (2.34).

The variables and are defined as and . For each term in the arrival time equations (2.27) and (2.28), an arc labeled or connects the departure time vertex of synchronizer to the arrival time vertex of synchronizer . Note that apart from representing each synchronizer with two vertices, the constraint graphs as defined thus far can be mapped directly to the circuit being modeled. The clock

\[
\begin{align*}
\Phi_p (F_i) &= F_i \\
\Phi_p (F_j) &= T_c + F_i \\
\emptyset (F_i; F_j) &= F_j - F_i \\
\emptyset (F_j; F_i) &= T_c + F_i - F_j
\end{align*}
\]

Figure 2.9: Time Shift Function Illustration

<table>
<thead>
<tr>
<th>Clock Constraints</th>
<th>Time Shift Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>((R_i - F_i) \mod T_c \geq w)</td>
<td>(\emptyset (e_j; e_i) = T_c - (e_j - e_i) \mod T_c)</td>
</tr>
<tr>
<td>((F_i - R_i) \mod T_c \geq w)</td>
<td>(2.24)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Early Signals</th>
<th>Late Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d_i = \min_{j \in I(i)} (d_j + \delta_{j,i} - \emptyset (L_j; L_i)))</td>
<td>(A_i = \max_{j \in I(i)} (D_j + \Delta_{j,i} - \emptyset (L_j; L_i)))</td>
</tr>
<tr>
<td>(a_i = H_i + Q_i)</td>
<td>(A_i \leq T_c - S_i + q_i)</td>
</tr>
<tr>
<td>latches: (d_i = \max (a_i + \delta_{D,i}, E_i' + q_i + \delta_{C,i}))</td>
<td>(D_i = \max (A_i + \Delta_{D,i}, E_i' + Q_i + \Delta_{C,i}))</td>
</tr>
<tr>
<td>flip-flops: (d_i = E_i' + q_i + \delta_{C,i})</td>
<td>(D_i = E_i' + Q_i + \Delta_{C,i})</td>
</tr>
</tbody>
</table>

Table 2.2: Timing Model Summary
system and its associated constraints are incorporated into the constraint graphs by adding two special vertices and sets of associated arcs. Clock distribution is modeled by connecting a source vertex to each latch departure time vertex. The source vertex is denoted by $S$ in the late signal constraint graph and $s$ in the early signal constraint graph. The arc weights are $E_i' + Q_i + \Delta_{C_i}$ in the late signal constraint graph and $E_i' + q_i + \delta_{C_i}$ in the early signal constraint graph. In both cases, the arc models the occurrence of the enabling clock event at synchronizer $s_i$. Arrival time constraints are modeled by connecting arrival time vertices to a sink vertex labelled $F$ or $f$ in the late and early signal constraint graphs, respectively. In the late signal constraint graph, these arcs represent setup time constraints and have weight $S_i - q_i - T_c$. In the early signal constraint graph, they represent hold time constraints and have weight $-(H_i + Q_i)$.

Shaded nodes in the graphs represent min functions; unshaded nodes correspond to max functions. The constraint graphs are defined so that in the late signal graph, $e(F) > 0$ if and only if a setup violation exists in the circuit, and so that $e(F)$, if positive, is the amount of the largest setup violation in the circuit. Similarly, in the early signal constraint graph, $e(f) < 0$ if and only if a hold violation exists in the circuit, and $-e(f)$, if positive, is the amount of the largest hold violation.

Primary inputs can be modeled with arcs from the source node to some latch arrival node $A_i$. The arc weights correspond to the early and late arrival times of the input signals shifted into the frame of reference of synchronizer $s_i$. Primary outputs require additional arcs that connect latch departure times to the sink node. The weights of these arcs are determined by the delay to the output pin and the required output time. Their values are again defined such that $e(F) = 0$ when the late signals arrive at their latest allowable times and $e(f) = 0$ when early signals arrive at their earliest allowable times.

Although the nodes and arrows have different labellings, the topologies of the late and early signal constraint graphs are identical. Also, both graphs contain cycles only when the circuit being analyzed contains feedback loops of level-sensitive latches. As discussed in Chapter III, when these graphs are acyclic, classical critical path techniques can be directly applied to the late
Figure 2.10: Graph Model of Timing Constraints
signal constraint graph to obtain critical paths that begin on the start node $S$ and end at the sink node $F$. Chapter III also discusses the more complicated analysis of the early signal constraint graph and of both graphs when they contain cycles. These graphs will also have a role in the analysis of optimal clocking methods in Chapter IV and the timing-driven design techniques of Chapter V.

2.3 Gate-Level Extension of the Timing Model

To effectively consider optimizations made at the gate level, we need to extend our timing model to allow for separate calculation of arrival and departure times associated with individual gates. This is straightforward, as illustrated in Figure 2.11 and Figure 2.12. Figure 2.11 illustrates a simple two-bit counter implemented using level-sensitive latches and a two-phase clock. The corresponding timing graph is shown in Figure 2.12. Latch-to-latch connections, which were once represented with single arrows, are now expressed as networks of nodes and arrows that reflect the gate-level structure of the circuit. Each node $G_i$ represents the output of a gate, and arrows fanning into these nodes are weighted with corresponding gate delays. Arrows connecting gate outputs to latch inputs $A_i$ are weighted with appropriate time shifts. The remaining arrows are unmarked; they are weighted similarly to those in Figure 2.10.

![Figure 2.11: Two-Bit Counter Circuit](image)

This expansion of the model allows us to express individual gate delays and possibly differing input-to-output delays through each gate, since each arc fanning into a gate node $G_i$ can be given a separate weight. However, it is now more difficult to model logic which mixes clock phases; i.e. logic in which gates combine signals that come from latches controlled by different
clock phases. This is because only a single time shift is associated with each arrow fanning into a latch input node \(A_i\). However, most latch designs use simple two-phase clocks where latches controlled by \(\Phi_1\) connect only to those controlled by \(\Phi_2\) and vice versa; these circuits, and many others, can be modeled with no restrictions.

![Expanded Constraint Graph for Counter Circuit](image)

Figure 2.12: Expanded Constraint Graph for Counter Circuit

### 2.4 Modeling Issues

This section elaborates our timing model by using it to discuss several issues in the timing of digital synchronous systems.

#### 2.4.1 Clock Skew Effects

An important assumption made by synchronous system designers is that clock signal events reach the appropriate synchronizing devices at the same time, e.g. a rising transition on clock signal \(\Phi_1\) should reach each synchronizer controlled by \(\Phi_1\) simultaneously. However, since there may be differing delays through the clock distribution network, clock signals arriving at different synchronizers may be skewed relative to one another.
Clock skew can have significant effects on circuit performance, particularly when cycle times are small. Its effects on system timing constraints may be seen using simple transformations of the synchronizer setup and hold time parameters. Assuming level-sensitive latches, combining the late signal departure time equation (2.32), the late signal propagation equations (2.28), and the setup time constraint (2.30) gives:

\[ E'_i + Q_i + \sum_{k=i}^{j-1} \Delta_{k,k+1} + \sum_{k=i}^{j-1} \varnothing (L_k, L_{k+1}) + S_j - T_c - q_j \leq 0 \]  

(2.35)

The clock skew terms \( Q_i \) and \( q_j \) can be lumped into an effective setup time \( S_{i,j}^* = S_j + Q_i - q_j \), simplifying (2.35) to the following:

\[ E'_i + \sum_{k=i}^{j-1} \Delta_{k,k+1} + \sum_{k=i}^{j-1} \varnothing (L_k, L_{k+1}) + S_{i,j}^* - T_c \leq 0 \]  

(2.36)

Note that this requires knowledge of latch \( l_i \), the latch at the start of the path. Clock skew terms can similarly be lumped into an effective hold time parameter, \( H_{i,j}^* \). Combining the early signal departure time equation (2.31), the early signal propagation equations (2.27), and the hold time constraint (2.29) gives:

\[ E'_i + q_i + \sum_{k=i}^{j-1} \delta_{k,k+1} + \sum_{k=i}^{j-1} \varnothing (L_k, L_{k+1}) - H_j - Q_j \geq 0 \]  

(2.37)

The effective hold time becomes \( H_{i,j}^* = H_j + Q_j - q_i \), and produces the following simplified constraint:

\[ E'_i + \sum_{k=i}^{j-1} \delta_{k,k+1} + \sum_{k=i}^{j-1} \varnothing (L_k, L_{k+1}) - H_{i,j}^* \geq 0 \]  

(2.38)

If we replace \( Q_i \) and \( q_j \) with global maximum and minimum clock propagation delays, \( Q \) and \( q \), the effective setup and hold times are increased by the difference \( Q - q \). From this, we see
that it is not the actual values of the clock delays that are significant, but their differences. A num-
ber of techniques have been developed which attempt to equalize delays in the clock distribution
network using balanced distribution trees and distributed buffers [4].

Several researchers have also observed that clock skew can be used to reduce the circuit
cycle time [4, 44, 46]. Recalling the definition of the effective setup time, \( S_{i,j}^* \), if \( Q_i < q_j \), then
\[ S_{i,j}^* < S_j \] and it thus may be possible to operate the circuit at a lower cycle time. This, however,
requires precise modeling and control of delays in the clock distribution network; it is usually
much easier to equalize these delays than to individually adjust them to particular values.

2.4.2 Latch Operating Modes

Since we are studying the timing behavior of synchronous systems, it is useful to step back
and address a fundamental question: what is the purpose of clocked circuit elements? At the tran-
sistor level, clocks can be used to precharge buses and the outputs of dynamic logic gates [51]. On
the other hand, at the gate level and above, there are two main reasons for clocking a circuit. First,
clocked structures allow us to synchronize the movement of data through a circuit by holding up
fast-moving early-arriving signals and preventing them from interfering with slower late-arriving
signals. Since it is impossible to perfectly control the delays in a circuit, there will always be some
variation in the delays along a path, and if no synchronization method were used, signals moving
along fast paths would eventually race forward and interfere with signals moving through slower
paths. One popular textbook notes that to run a computer with a 200ns cycle time asynchronously
and error-free for a year, the delays within the computer would need to be known within an accu-
"racy of less than \( 6 \times 10^{-22} \) seconds [51, p. 331]!

Recent work in the area of wave pipelining [142] (originally known as maximum-rate
pipelining [29]) has made it possible to reduce the number of clocked elements in a circuit and still
guarantee error-free operation. In wave-pipelined designs, synchronizing elements are removed or
omitted whenever the difference between the minimum and maximum delays between a pair of
synchronizers can be sufficiently bounded to ensure that successive waves of data moving through
the combinational logic will not interfere with one another. Synchronizing elements are still used,
but in a minimal number of places. This eliminates the area and delay overhead of these devices while still ensuring correct circuit operation. Wave pipelining is discussed in more detail in Section 2.4.3.

The second reason for using clocked devices is that they may be used as storage elements in a circuit. When present, they preserve the internal state of a stopped machine allowing it to be observed for testing purposes or to be used when the machine is restarted. This is an important application, and can be used to improve the testability of wave-pipelined circuits.

Because level-sensitive latches allow signals to begin propagating through the latch as soon as they arrive, it is possible for a level-sensitive latch to be used as a storage element without performing any synchronization. To clarify this distinction, we can classify latches as either synchronizing or non-synchronizing. A latch is said to be synchronizing if it serves to hold up an early-arriving signal at the latch, or in the notation of Section 2.2.2, if \( a_i + \delta_{Di} < E'_{i} + q_i + \delta_{Ci} \). A latch is non-synchronizing if the converse is true; if \( a_i + \delta_{Di} \geq E'_{i} + q_i + \delta_{Ci} \). Observe that because it does nothing to hold up signal movement, a non-synchronizing latch has no effect on circuit operation (other than to introduce an extra latch delay) and that it may be replaced by a delay line having equivalent delays. However, non-synchronizing latches can continue to be used as storage registers which can allow a machine to be stopped without loss of state information. Figure 2.13 illustrates the distinction between synchronizing and non-synchronizing latches, where for simplicity, clock skew and latch delay terms are omitted. Note that the synchronization properties of a latch are dependent upon system timing; increasing the cycle time can cause non-synchronizing latches to become synchronizing. In general, a more optimal clock schedule can be found when

![Figure 2.13: Example Timings of Synchronizing and Non-Synchronizing Latches](image-url)
latches are allowed to be non-synchronizing than when they are assumed to all be synchronizing. This will be discussed in more detail in Chapter IV.

We may also make a distinction in latch operation based on the late arrival times to the latch. We say that a latch is fully synchronizing if \( A_i + \Delta_{Dj} < E_i' + Q_i + \Delta_{Cj} \). A latch is partially synchronizing if \( A_i + \delta_{Dj} < E_i' + q_i + \delta_{Cj} \) but \( A_i + \Delta_{Dj} \geq E_i' + Q_i + \Delta_{Cj} \). These distinctions are illustrated in Figure 2.13; for simplicity, clock and latch delays are assumed zero.

Edge-triggered flip-flops (with nonzero setup times) do not allow signals to flow directly through them as latches do; as a result, we consider all flip-flops in a circuit to be fully synchronizing.

### 2.4.3 Wave Pipelining

As mentioned in the previous section, if the difference between the minimum and maximum delays through a combinational block is sufficiently bounded to prevent interference between successive waves of data, more than one wave can be propagated through the block simultaneously. This mode of operation was first described in 1969 by Cotten [29] and was originally called maximum-rate pipelining. Interest in this work has been recently revived under the new name wave pipelining [142]. An example of wave-pipelined operation is shown in Figure 2.14. The storage elements in the circuit are negative edge-triggered and the delay through the logic connecting flip-flop \( f_i \) to flip-flop \( f_j \) has a minimum value of 10ns and a maximum value of 15ns. The circuit is clocked by a two-phase overlapping system of clocks for which \( E_i = L_i = \frac{1}{4} T_c \) and \( E_j = L_j = T_c \). In the timing diagram for the first case shown, signals departing on the falling edge of \( \Phi_{p_i} \) are required to arrive at the input of \( f_j \) by the first falling edge of \( \Phi_{p_j} \), resulting in a minimum cycle time of \( 15ns \times \frac{4}{3} = 20ns \). The shaded triangles in the timing diagrams illustrate the flow of data through the logic connecting \( f_i \) to \( f_j \). They illustrate the fact that the fastest-moving signals reach the block outputs 5ns before the slowest signals arrive. In case 2, signals propagating from \( f_i \) to \( f_j \) are not to required to reach \( f_j \) until the second falling edge of \( \Phi_{p_j} \). But since a new
wave of data enters the block on every cycle, we now have portions of the clock cycle during which two waves of data are propagating through the block simultaneously. Observe that during these intervals, it is impossible to stop the clock system, leaving the clock signals at their current values, without losing one of the two waves of data moving through the wave-pipelined circuit. If we stop the clocks at any time in the interval \( \left[ \frac{1}{4} T_c, T_c \right] \), the newest wave of data in the block will rush forward and overwrite the older wave before it can be stored in \( f_j \). However, we can safely stop the clocks during the interval \( \left[ 0, \frac{1}{4} T_c \right] \) since only one wave of data is propagating during this time. Note that the cycle time is reduced to \( 15 \text{ns} \times \frac{4}{7} = 8.57 \text{ns} \). For Case 3, signals...
departing from \( f_i \) are not required at the input of \( f_j \) until the third falling edge of \( \Phi_{p_j} \) after they have departed. This allows the cycle time to be reduced further to \( 15\text{ns} \times \frac{4}{11} = 5.45\text{ns} \). Now there are portions of the clock cycle during which three waves of data are propagating through the block connecting \( f_i \) to \( f_j \) and there is no portion of the clock cycle during which a single wave of data is moving through the path. As a result, there is now no interval of time during which the circuit may be stopped without losing one or more waves of data moving through the wave-pipelined path.

As originally stated by Cotten [29], the minimum cycle time of a wave-pipelined circuit is limited only by the difference between the maximum and minimum delays through the wave-pipelined logic and not by the actual maximum delay of the circuits. This is evident in the example just discussed. If we attempt to add an additional degree of wave-pipelining, the cycle time would be reduced to \( 15\text{ns} \times \frac{4}{15} = 4\text{ns} \). However, the difference between the maximum and minimum delays for this path is 5ns, and from looking at Case 3 in Figure 2.14 we see that reducing the cycle time below this difference would cause successive waves of data to interfere with one another, causing errors in circuit operation.

Unfortunately, simply equalizing the delays in a circuit will not always ensure that the circuit can be wave-pipelined. Instead, we must also consider the granularity of the circuit we are wave-pipelining. If the delays in our previous example had been due to a single typical logic gate, then we could not expect to send more than one signal through this gate simultaneously. If a wave produced a different output than its predecessor, then the predecessor’s output signal would be degraded or destroyed.

Joy and Ciesielski [72] suggest a method for guaranteeing signal separation in wave-pipelined circuits. They require that for each gate in a circuit, no changes can be made to the gate’s inputs until the outputs of all of its fanout gates have stabilized. While this perhaps is sufficient to guarantee correct operation, it may be overly restrictive. Instead, it should be sufficient to require that every gate’s output stabilize before its inputs change. Either case requires a very large number of extra constraints: \( O(n^2) \), where \( n \) is the number of inputs to the wave-pipelined block.
If we know that all signals will be adequately separated, we can easily accommodate wave-pipelined paths in our timing model by adding $nT_c$ to the time shift function $\emptyset (L_j, L_i)$, where $n$ is the number of additional cycles to wait before expecting the wave to arrive at synchronizer $s_i$.

### 2.4.3.1 Wave-Pipelining Due to Clock Skew

Wave pipelining effects can arise unintentionally as a result of clock skew. For example, consider the circuit section shown in Figure 2.15. The circuit is controlled with edge-triggered flip-flops, and the triangles in the timing diagram indicate the flow of signals through the combinational logic between flip-flops. The timing diagram on the left is for the circuit free of clock skew; the one on the right describes the case when the clock to $f_j$ is delayed by 2ns. Without clock skew, there is no wave pipelining in the circuit; the path carries exactly one signal between flip-flops. However, when the indicated clock skew is added, wave pipelining is observed in $C_{i,j}$ during the lightly-shaded intervals. This is despite the fact that signals are already stable on the input of $f_j$; since the old signal has not yet been latched when a new signal begins propagating, the path is wave-pipelined. In the case shown, clock skew has eased the setup constraint and may allow the clock cycle time to be reduced; such reductions have been observed and studied by other researchers [44, 46].

Because of this additional wave-pipelining, whenever clock skew is possible in a circuit, we must be sure that the minimum delays are sufficient to support the wave pipelining which can be unintentionally introduced and that no signal collisions occur. From Figure 2.15, if the clock skew is small compared to the other delays in the circuit, then wave pipelining effects will be small and can be masked by extra margin in synchronizer setup and hold times. However, as clock skew grows in significance, it will become increasingly important to be aware of these unintentional wave pipelining effects.
For certain clock schedules, wave-pipelining can arise as a consequence of normal latch operation. For example, the circuit in Figure 2.16-a is clocked with a single-phase clock and uses level-sensitive latches. Assuming that the minimum delays are sufficiently large to allow correct level-sensitive operation, the path from $l_i$ to $l_j$ is essentially wave-pipelined, as two waves of data are allowed to propagate through $C_{i,j}$ while the clock $\Phi_1$ is high. As a result, we must also consider the possibility of collisions between these signals within $C_{i,j}$ and must enforce the constraints of Joy and Ciesielski.

Figure 2.16-b illustrates the general condition which must be satisfied in order for this form of wave-pipelining to arise. Denoting the length of the time latch $l_i$ is open as $T_i$, wave-pipelining will arise on a path from a positive level-sensitive latch $l_i$ to a synchronizer $s_j$ (of arbitrary type) if and only if:

$$T_i + \varnothing(L_i, L_j) > T_c$$  \hspace{1cm} (2.39)
Proof of this follows immediately from the timing diagram in Figure 2.16-b. Wave-pipelining occurs when a new signal begins propagating along a path before the previous signal in the path is latched. In a periodic system, new signals begin propagating every $T_c$ time units, and $T_i + \mathcal{O}(L_i, L_j)$ is the amount of time between when a signal begins propagating from a latch and when it is latched into the device at the end of the path.

We define *single-phase clocking* as the clocking of a circuit with a single clock phase and where latches are either all positive level-sensitive or all negative level-sensitive. Clocking schemes which mix latch types but still use a single signal are considered multi-phase. By definition, single-phase systems have a time shift of $\mathcal{O}(L_i, L_j) = T_c$ and since for latches, $T_i \geq w > 0$, in single-phase systems, wave-pipelining will *always* occur in the combinational segments that follow level-sensitive latches.

**2.4.4 Stoppable and Restartable Machines**

As mentioned in the previous section, it is possible to build and operate a circuit which exhibits correct steady-state behavior but which cannot be stopped without loss of information about the machine’s state. If state information can be preserved when the clock signals are stopped, then we say that the circuit is *stoppable*. To be stoppable, a circuit must contain at least one storage element for each piece of data; this may require the use of the *keeper* registers mentioned in Sec-
tion 2.4.2. We also note that if level-sensitive latches are used, the latch used to store each piece of data should be switched off (closed) when the circuit is stopped.

A circuit is called restartable if it can be started without loss of information due to timing errors. For a newly-started circuit, all signals depart on the first enabling edge for each device. In order to guarantee that no timing errors occur, we must ensure that signals which depart on these enabling edges do not cause hold violations in any later stages of the circuit. Another way to phrase this is that for a circuit to be restartable under a specified clock schedule, all hold constraints must be satisfiable even when all latches are synchronizing (the worst case). This will have important ramifications on the optimal clocking procedures described in Chapter IV.

If a circuit is both stoppable and restartable, then we can say that it is steppable, meaning that it can be single-stepped without error, an important property for testability.

2.4.5 Inputs, Outputs, and Hierarchical Models

Our approach to handling inputs and outputs is based on the desire to model circuits hierarchically. In general, circuits are modeled as sets of interconnected components, with the timing behavior of each component described in terms of a macromodel. At the lowest levels, these components can be simple synchronizing devices, such as D-type flip-flops or latches, or they can be blocks of purely combinational logic. The macromodels for these devices have already been discussed, and consist of two parts: (optional) input constraints, which describe when input signals must arrive to ensure correct operation, and input-output equations, which describe when output signals can change as a function of when device inputs change.

At higher levels of abstraction, we can also write timing macromodels that describe the timing behavior of subsystems. An interconnected pair of such subsystems is shown in Figure 2.1. As with the primitive components, macromodels for subcircuits have two parts: input constraints and input-output equations. These macromodels can be derived from the internal structure of a subcircuit (bottom-up macromodeling), or they can be specified prior to the subcircuit design by a top-down design method.

When used in a bottom-up design methodology, the timing macromodel of a subsystem represents a set of constraints imposed on the higher-level system by the already-designed sub-
These constraints are predetermined and unalterable. The higher-level system must be designed to accommodate them and ensure that all subsystems are used in a timing-correct manner. The macromodels for primitive devices are examples of models used for bottom-up design. The input constraints describe when input signals are allowed to change; the input-output relations describe when the output signals are going to change, as a function of circuit inputs.

Top-down designs impose constraints in the opposite direction: a higher-level system imposes constraints on yet-to-be-designed subsystems. This requires macromodels having the same two components as already described, but with slightly different interpretations. Now the input constraints describe when inputs are going to change and the subcircuit must be guaranteed to operate correctly over this full range of input times. The input-output relations describe when the subcircuit output signals are allowed to change; to ensure correct interaction with its environment, a subcircuit must guarantee that its outputs only change within the specified interval.

Figure 2.19 illustrates how ports can be used to express input and input-output constraints for both bottom-up and top-down timing analysis. Although only one input and one output port is

Figure 2.17: Timing Model for Interacting Machines
shown in the figure, an arbitrary number of such ports can be used, one for each input or output signal. In a bottom-up analysis, the internal structure of a subcircuit is used to determine when inputs to the structure are allowed to change, i.e. within the interval \((a_{in}, A_{in})\). The range of possible output times is also determined from the subcircuit structure, and is indicated by the pair \((d_{out}, D_{out})\). Note that \((d_{out}, D_{out})\) can be a function of the actual arrival times at the subcircuit.

In a top-down analysis, the specification of the external environment determines the range of times during which inputs to the block will change. This range of times is indicated by the \((d_{in}, D_{in})\) pair. Restrictions on when subcircuit outputs can be allowed to change are described by \((a_{out}, A_{out})\).

In all cases, it is necessary to indicate a frame-of-reference within which the specified times are defined. This frame-of-reference can either be the global reference frame or the frame-of-reference of some specified clock. The arrival and departure times for each port need not be specified within the same reference frame.
2.5 Conclusions

In this chapter, we have developed and described a basic model for the timing of circuits with level-sensitive latches. The model is an extension of the one developed by Sakallah, Mudge, and Olukotun [105, 114, 116] and allows for clock skew and arbitrary combinations of positive and negative edge-triggered and level-sensitive devices. The following chapters rely heavily upon this model. The graph formulation presented in Section 2.2.3 is of special interest, as Chapters III and V use it as a basis of an extension of the Critical Path Method to circuits with level-sensitive latches.

To illustrate the power of the model, we also used it to describe a number of important timing-related concepts. We discussed the effects of clock skew on system operation and the function of latches and flip-flops as synchronizing elements. We also considered wave-pipelining and cases where wave-pipelining could arise unintentionally. We described constraints which would allow a circuit to be easily stopped and started, and finally explained our input and output timing constraints in light of a hierarchical modeling strategy.
CHAPTER III
TIMING VERIFICATION

3.1 The Timing Verification Problem

The timing verification problem for sequential circuits is to determine whether a sequential circuit with a specified structure will operate correctly under a desired clock schedule. To do this successfully, it is necessary to have accurate models for the timing behavior of each component in the circuit. To verify that a circuit will run correctly under a given clock schedule, all possible paths through the circuit must be analyzed to guarantee that at no point will the interacting clock and data signals produce undesired behavior.

Few designers, however, are satisfied with a simple determination of whether or not a circuit will meet its timing constraints. If the design fails to meet its requirements, it may be useful to know how close it came to its desired performance. This falls into the domain of optimal clocking methods [114, 132, 120], which find the minimum cycle time for a circuit; but these still provide no information about why a circuit fails to meet its desired timing. To determine this, it is necessary to know the critical paths in a circuit, those sections of the circuit that place the tightest constraints on its timing behavior. The concept of a critical path has been used for many years to analyze both combinational and sequential circuits [79, 59] and underlies a large number of timing-driven optimization techniques. However, these approaches assume that only edge-triggered storage elements are used, leading to a very simple set of timing constraints. If a circuit instead uses level-sensitive latches, the corresponding timing constraints are complicated by the transparent behavior of enabled latches. Designers are forced to make restricting assumptions to allow use
of existing critical path-based optimization techniques; most commonly, the transparent latch properties are ignored and latches are conservatively treated as edge-triggered devices.

This chapter extends the classical definition of a critical path to fully account for the timing properties of level-sensitive latches. We show that three distinct types of critical paths can arise in contrast to the single type of critical path commonly observed in edge-triggered circuits. Each path type is described in detail and in a framework which relates them to classical critical path methods. We then discuss procedures for extracting these paths and analyze the factors which affect the feasibility of extraction and enumeration. We present two approaches, one based on existing relaxation timing verification methods and one a new algorithm which iteratively constructs possible critical paths. Finally, we analyze the algorithmic complexities of each approach and discuss their application.

We review a simple timing verification procedure in Section 3.2. Section 3.3 presents the three types of critical paths that can arise and Section 3.4 describes approaches for identifying these paths during and after timing verification and concludes with the results of experiments performed using the various verification and path identification procedures.

3.2 A Simple Verification Algorithm

The verification method we consider is that of simply relaxing, or iteratively resolving, the timing constraints until a fixed-point solution is found. The relations in Table 2.2 can be classified into two types: equations that describe circuit operation and constraints that reflect underlying assumptions. The equations can be used to calculate the values of the arrival \( (a_i, A_i) \) and departure \( (d_i, D_i) \) times, subject to the assumption that all synchronizers are operating correctly. The constraints are then used to verify that these times fall within the required ranges in order for the equations to be valid. It was shown by Szymanski and Shenoy that the arrival and departure time equations can be solved by relaxation until a fixed point, or stable solution, is obtained [133]. A commonly-used procedure is shown in Figure 3.1 (Algorithm SIMPLE-RELAX). If the arrival times are initialized to \( a_i = A_i = -\infty \), then this relaxation algorithm simulates the start-up timing of the circuit, with each iteration simulating the events of one clock cycle. Szymanski and Shenoy also observed that if multiple solutions exist for the arrival and departure time equations, these ini-
for all latches
    initialize arrival times $a[i]$, $A[i] = -\text{INFINITY}$
end for
repeat
    for all latches
        $D[i] = \max (A[i] + \text{DeltaD}[i], E'[i] + Q[i] + \text{DeltaC}[i])$
        $d[i] = \max (a[i] + \text{deltaD}[i], E'[i] + q[i] + \text{deltaC}[i])$
    end for
    for all latches
        $A[i] = \max_j (D[j] + \text{Delta}[i,j] - \text{PHI}[L_i,L_j])$
        $a[i] = \min_j (d[j] + \text{delta}[i,j] - \text{PHI}[L_i,L_j])$
    end for
    until no times have changed or a maximum number of iteration
    for all latches
        check setup constraint: $A[i] \leq Tc - S[i] + q[i]$
        check hold constraint: $a[i] \geq H[i] + Q[i]$
    end for

Figure 3.1: Algorithm SIMPLE-RELAX: Relaxation Verification Algorithm

Torial values ensure that the most physically meaningful solution is found, as they result in the solution with the lowest possible arrival time values.

A procedure similar to Algorithm SIMPLE-RELAX was used by Sakallah, Mudge, and Olukotun in the checkTc program [116]. The first performance bounds on the relaxation verification approach were reported by Ishii et. al. [65] and later by Szymanski and Shenoy [133]; both groups observed that relaxation of the equations corresponds to a Bellman-Ford solution of a simple set of linear constraints. This terminates in at worst $O(|L||E|)$ time, where $|L|$ is the number of latches in the circuit and $|E|$ is the number of edges, or latch-to-latch connections. If a solution exists, it will be found in at most $|L|$ iterations, so that $|L|$ should be used as the iteration limit in Figure 3.1. Each iteration involves examining up to $|E|$ edges. Since $|E|$ is at most $|L|^2$, the worst-case performance of the relaxation algorithm is cubic in the number of latches.

The worst-case performance of the algorithm occurs when there is a positive-weight cycle in the constraint graphs of Figure 2.10. When such a cycle is present, the arrival and departure time equations have no feasible solution. In each iteration, the arrival and departure times will steadily increase without bound. Since it is an instance of the Bellman-Ford algorithm, Algorithm SIM-

PLACE-RELAX requires $|L|$ iterations to determine whether the arrival and departure times diverge. However, as shown in Figure 3.2, we can often identify that a timing problem exists much earlier.
Embedded in a larger circuit, the loop shown has a delay slightly larger than the cycle time allows. Since no solution exists, Algorithm SIMPLE-RELAX will detect an error after \(|L|\) iterations; but we can identify that a problem exists in the 11th iteration, when the arrival time at latch \(l_2\) becomes large enough to cause a setup violation. How soon this happens in general depends on the size of the loop violation. Observe that the arrival and departure times increase by the amount of violation every \(|P_{\text{loop}}|\) iterations, where \(|P_{\text{loop}}|\) is the size of the loop. This is the expected worst-case (slowest) rate of increase for the relaxation algorithms when verifying violated loops, as shown by the following analysis:

Each update of an arrival time will have the form: 
\[
A_i \leftarrow D_i - 1 + \Delta_i - 1,i - \varnothing (L_i - 1, L_i). 
\]

When all of the latches are transparent, for each latch in the loop, \(D_i = A_i + \Delta_{D_i}\), and if the loop is critical, then for all of the latches except the one being updated,
\[
A_i = D_i - 1 + \Delta_i - 1,i - \varnothing (L_i - 1, L_i). 
\]

Combining these equations gives
\[
A_i \leftarrow A_i + \Delta (P_{\text{loop}}) - \varnothing (P_{\text{loop}}), \text{ where } \Delta (P_{\text{loop}}) \text{ and } \varnothing (P_{\text{loop}}) \text{ are the total delay and time shift around the loop, respectively:}
\]

\[
\Delta (P_{\text{loop}}) = \sum_{j \in P_{\text{loop}}} (\Delta_{D_j - 1} + \Delta_{j - 1,j}) \tag{3.1}
\]

\[
\varnothing (P_{\text{loop}}) = \sum_{j \in P_{\text{loop}}} \varnothing (L_{j - 1, j}) \tag{3.2}
\]

The amount of increase in each update is \(V = \Delta (P_{\text{loop}}) - \varnothing (P_{\text{loop}})\), the amount of the loop violation. In the worst case, only one latch is updated in each iteration, so that each arrival time will increase by \(V\) every \(|P_{\text{loop}}|\) iterations.

We can stop the unbounded upward growth of arrival and departure times by using the original latch model illustrated in Figure 2.8. The simplified model presented in Table 2.2 allowed signals to depart at any time after the latch’s enabling event, even after the latch was re-closed. If instead we use the model: 
\[
D_i = \min (\max (A_i + \Delta_{D_i}, E_i, Q_i, \Delta_{C_i}), L_i + O_i + \Delta_{C_i}), \text{ signals can only leave the latch in the interval between the device’s enabling and latching events. For each}
\]
latch, this adds a min node to the late signal constraint graphs illustrated in Figure 2.10. The modified graph structure is shown in Figure 3.3-a. Late departure times are thus clipped to occur no later than the device’s latching event.

Clipping was originally used by Sakallah, Mudge, and Olukotun in the check\(T_c\) program [116]. The introduction of the min function does not eliminate the possibility of positive-weight cycles existing in the graph, but it does prevent them from causing variables to increase without bound. When a positive-weight cycle exists, the min nodes “short-circuit” it, effectively breaking the cycle when it no longer produces the smallest input to the min node.

In most cases, this will cause the relaxation to stop after fewer iterations, but if the amount of violation, \(V\), is small, a full \(|L|\) iterations may still be required to discover that a loop constraint is violated. This is a result of the fact that we cannot clip a departure time until it exceeds \(L'_i + Q_i\) and that in the worst case, the departure time at each node in a critical loop will increase by the amount of violation every \(|P_{\text{loop}}|\) iterations, where \(|P_{\text{loop}}|\) is the size of the loop. This effect is demonstrated in Section 3.5, where we demonstrate algorithm CLIP-RELAX, a version of algorithm SIMPLE-RELAX which has been augmented to clip non-physical departure times.

### 3.3 Critical Paths in Circuits with Level-Sensitive Latches

The Critical Path Method was described in Section 1.2.1.1, and Section 1.2.3 described a number of circuit optimizations based on critical path information. Each of these techniques...
assume that circuits are controlled with edge-triggered flip-flops and that critical paths can only exist in the acyclic combinational logic that lies between flip-flops or circuit inputs and outputs.

When circuits are instead clocked with level-sensitive latches, the concept of a critical path becomes more complicated. Since signals can flow directly through level-sensitive latches, critical paths can now extend through latches and can include both combinational and sequential circuit elements. For example, in the circuit shown in Figure 3.4, the setup violation at latch $l_3$ can be eliminated by reducing either of the delays $\Delta_{1,2}$ or $\Delta_{2,3}$. Thus both of these delays should be considered part of the critical path from latch $l_1$ to latch $l_3$. The definition of a path must, therefore, be extended to include multiple combinational segments joined by transparent latches.

This extension complicates the analysis in two ways. The first complication appears when we analyze short (minimum delay) paths in latched circuits. For short paths, the constraint graph is complicated by the presence of nodes that correspond to both max and min functions. Existing CPM methods do not provide for such mixed-node graphs. The second difficulty is that the late and early signal constraint graphs may contain cycles. If a circuit contains feedback, then the
project networks will contain cycles, and some of these cycles can constrain circuit timing. Classical CPM approaches are unable to deal with cyclic project networks.

In this section we formally define three types of critical paths: critical long paths, critical short paths, and critical loops. All of these types can be observed in the late and early signal constraint graphs and are presented in increasing order of complexity. We also discuss these critical paths directly in terms of latches and combinational logic blocks. In this context, a path \( P \) is a sequence of latches \( P = l_0 \rightarrow l_1 \rightarrow \ldots \rightarrow l_m \) where each latch is directly connected to its predecessor through a combinational logic segment (Figure 3.5). In the following sections we assume, without loss of generality, that the latches in the path are numbered from 0 to \( m \). The length of \( P \) is defined as the number of combinational segments in the path so that \( |P| = m \). Path delays \( \delta(P) \) and \( \Delta(P) \) are defined as the sum of the minimum and maximum delays along the path, respectively, and the path latency \( \Lambda(P) \) is defined to be the number of clock cycles available for signals to propagate the length of the path. For an acyclic path \( P \), latency is related to time shifts, cycle time, and the enabling events by the following equation:

\[
\Lambda(P) = \frac{1}{T_c} \left( \sum_{i=1}^{m} \Delta(L_{i-1}, L_i) + T_c - E'_0 \right)
\]  

(3.3)

For the verification problem, \( \Lambda(P) \) is constant since the clock schedule is fixed. \( \Lambda(P) \) is also constant when all clock event times scale uniformly with the cycle time, \( T_c \).

These definitions allow for the existence of parallel critical paths. When there are multiple critical paths fanning into a latch \( l \), they can be viewed as a tree rooted at latch \( l \). The width of a path is then defined to be the number of leaf nodes in the tree. Also, it has been assumed that we are only considering a single critical path through each combinational segment between latches. If
parallel combinational subpaths are present, the total number of paths grows combinatorially. Fortunately, in most cases these paths can be represented implicitly.

Several other researchers have independently developed similar critical path definitions. Ishii and Leiserson [65] describe critical paths in the context of the *computational expansion* of a latch-controlled circuit; this is essentially an unrolled version of a cyclic circuit. A more closely related definition of critical long paths was developed by Lockyear and Ebeling [89] (and Ishii et al. [66]) for use in retiming applications, and more recently, Shenoy et al. [122] used a definition similar to ours to allocate slack among stages of pipelined circuits.

Our approach is unique in that we relate these paths directly to classical critical path-based approaches, suggesting the extension of a wide variety of existing timing-driven design approaches to circuits with level-sensitive latches. We also consider the effects of clock skew in our formulation and consider practical issues in the reporting and application of critical path information. The three types of critical paths that can arise are presented in the following three subsections.

### 3.3.1 Critical Long Paths

The first type of critical path we describe is called a *critical long path* and results from the setup constraint at the input of each latch. Such a path corresponds to a critical path in the late signal graph that connects the source and sink nodes, as shown in Figure 3.6-a. Actual and required times are defined as in Section 1.2.1.1, although their calculation is complicated by the possible presence of cycles in the late signal constraint graph. Procedures for calculating these times are presented in Section 3.4.1, allowing us to present the formal definition of a critical long path as follows:

**Definition 3.1** A critical long path is a path in a late signal constraint graph consisting of a cycle-free sequence of critical arcs which connect the source and sink nodes.

Critical arcs are defined as in Section 1.2.1.1 as the arcs in the project network having the smallest or most negative slack. Comparing this definition with the late signal graph of Figure 2.10-a, we see that the following set of constraints must hold for any critical long path:

\[
D_0 = E'_0 + Q_0 + \Delta_0^1
\] (3.4)
\[ \forall i \in \{1, \ldots, m\}, A_i = D_{i-1} + \Delta_{i-1, i} - \emptyset (L_{i-1}, L_i) = D_{i-1} + \hat{\Delta}_{i-1, i} \quad (3.5) \]

\[ \forall i \in \{1, \ldots, m-1\}, D_i = A_i + \Delta_i \quad (3.6) \]

\[ \forall i \in \{1, \ldots, \lfloor L \rfloor\}, A_m - (T_c + q_m - S_m) \geq A_i - (T_c + q_i - S_i) \quad (3.7) \]

\( |L| \) is the number of latches in the circuit. Critical long paths are required to be acyclic, although they will exist in both cyclic and acyclic circuits. Cyclic critical paths are described in Section 3.3.3, and in Section 3.4.2 it is shown that we can safely decouple the analysis of loops in cyclic constraint graphs.

Similar definitions of critical long paths were presented previously by Ishii, Leiserson, and Papaelthymiou [65, 66] and Lockyear and Ebeling [89]; both groups of researchers were primarily interested in retiming optimization problems; their definitions of a critical long path included all paths which could constrain the level-sensitive retiming problem. Both definitions implied a set of \(|V|^2 \) critical paths, where \(|V| \) is the number of combinational blocks (gates) in a circuit. In contrast, we focus on the path (or paths) which correspond to the largest setup time violation in the circuit, as these paths correspond to the largest delay reduction necessary to allow the circuit to operate at the desired speed. “Sub-”critical paths can then be identified using slack and float information from the constraint graphs.

The following theorem describes the relationship between critical long paths and the setup time constraints in a circuit:

\[ \text{Figure 3.6: Sample Critical Long Path} \]

1. Note that, for simplicity, we assume a single delay through each latch, \( \Delta_i \).
Theorem 3.1. A path \( P \) is a critical long path if and only if the following two conditions are satisfied: (L1) any increase in a delay along the path will tighten or worsen the most severe setup time constraint in the circuit, and (L2) a decrease to any delay in the path will reduce the severity of this constraint, as long as no parallel critical paths block the change.

Proof: (only if part) It is simple to show that if \( P \) is a critical long path, conditions (L1) and (L2) will be satisfied. Recall that in the late signal graphs, the event time at the sink node, \( e (F) \), is the amount of the largest setup violation in the circuit. If \( e (F) \) is negative, then it represents the amount of slack in the tightest setup constraint. If \( P \) is a critical long path, then equations (3.4)-(3.7) guarantee that \( P \) determines the value of \( e (F) \), which can be calculated by simply summing the arc weights along \( P \). An increase in any delay along \( P \) will thus cause \( e (F) \) to increase. Reducing any delay in \( P \) will allow \( e (F) \) to be reduced, but \( e (F) \) will only be reduced if there are no other critical paths in parallel with \( P \). This is because the event time at each node is the maximum of its input event times: increasing the maximum input time will always cause a change on the output; reducing the maximum input time will only cause a change when the reduced time remains the maximum input time.

(if part) We also can show that if conditions (L1) and (L2) are satisfied, \( P \) is a critical long path. Except for the qualification for parallel paths made in condition (2), both conditions require that the sensitivity of the tightest setup constraint to changes in \( P \) be nonzero. This implies that the event time at the sink node, \( e (F) \), be equal to the sum of the arc weights along \( P \), a condition which can only be true if equations (3.4)-(3.7) are satisfied for \( P \). These equations imply that the slacks on each arc of \( P \) equal the smallest slack in the network; therefore, \( P \) must be a critical long path. □

The timing of a typical critical long path is illustrated in Figure 3.6-b. Examining this, we make the following remark:

- Combining conditions (3.4)-(3.7), we see that a critical long path constraint with zero slack is of the form:

\[
\sum_{i=1}^{m} \left( \Delta_{i-1} + \Delta_{i-1,i} \right) + S_{m} + Q_{0} - q_{m} = \sum_{i=1}^{m} \left( D_{i-1} , L_{j} \right) + \left( T_{c} - E^*_{0} \right) = \lambda (P) T_{c} (3.8)
\]
The terms on the leftmost side of the equation represent times required for data to propagate through logic and get set up at the input of the final storage device. The terms in the center and on the right represent the time available for these operations to take place. If the setup constraint were violated, the leftmost “=” would be replaced by “>”.

3.3.2 Critical Short Paths

The second type of critical path is called a critical short path. It results from the hold constraint at each latch input and corresponds to critical paths in the early signal graphs as shown in Figure 3.7-a. The formal definition of a critical short path is as follows:

**Definition 3.2** A critical short path is a path in an early signal constraint graph consisting of a cycle-free sequence of critical arcs which connect the source and sink nodes.

Critical arcs are those arcs having the smallest or most negative slack, where slack is now defined for the early signals, as in equations (1.7) and (1.8). However, since the early signal constraint graphs contain both min and max nodes, we must extend the definitions of actual and required times given in Section 1.2.1.1. We can easily extend the event time definition using equation (1.1) to specify event times of max nodes and equation (1.5) for event times of min nodes [37]. The required time definition is more complex; we first illustrate it for the general mixed min/max graphs shown in Figure 3.8; it can then be easily applied to the early signal constraint graphs.

Figure 3.7-a shows a graph containing predominantly max nodes where the sink node is also a max node. Actual and required event times are shown next to each node. Actual event times are easily determined, but the required time calculation is complicated by the presence of the min
node B. Since the sink node is a max node, required times in Figure 3.8-a are the latest times that events can occur without increasing the project completion time. The required time for node F is 6, and using equation (1.4), the required time for B is \( 6 - 5 = 1 \). However, when we consider event A, we see that the time of event A could be made arbitrarily late without increasing the time of event B or the completion time at the sink node. The required time for event A is effectively infinite. A similar analysis for Figure 3.8-b determines that the required time for event a is \(-\infty\).

Thus we define the following general rules for determining required times in mixed min/max CPM graphs:

1. The type of the graph is determined by the type of the sink node. If the sink node computes a max function, the graph is called a max graph. If the sink node computes a min function, the graph is called a min graph.

2. Max nodes in min graphs and min nodes in max graphs are termed minority nodes.

3. The required times for all nodes are determined using equations (1.4) and (1.8), but with the following modification: if an output of a node \( u \) does not determine the output of a minority node \( v \), i.e., \( e(v) \neq e(u) + r(\text{e.v}) \), then a minority required time, \( r'(v) \) is used to calculate \( r(u) \).

4. In max graphs, the minority required times are \( r'(v) = \infty \). In min graphs, the minority required times are \( r'(v) = -\infty \).

These rules allow required event times to be defined for each node in a mixed min/max CPM graph. Slacks and floats are defined as before, and a critical path can again be identified as a path from the source to the sink consisting of arrows with the most negative float. In the degenerate case where more than one node determines the value of a minority node, all such nodes are assigned non-infinite slacks and can be potentially part of a critical path.
Examining the graph of Figure 3.7-a, we see that the following constraints must hold for any critical short path:

\[ d_0 = E'_0 + q_0 + \delta_0 \] (3.9)

\[ \forall i \in \{1, \ldots, m\}, \ a_i = d_{i-1} + \delta_{i-1,i} - \delta (L_{i-1}, L_i) = d_{i-1} + \hat{\delta}_{i-1,i} \] (3.10)

\[ \forall i \in \{1, \ldots, m-1\}, \ d_i = a_i + \delta_{i-1} \] (3.11)

\[ \forall i \in \{1, \ldots, |L|\}, \ (H_m - Q_m) - a_m \geq (H_i - Q_i) - a_i \] (3.12)

The following theorem describes the relationship between critical short paths and the hold time constraints in a circuit:

**Theorem 3.2.** A path P is a critical short path if and only if the following two conditions are satisfied: (S1) any decrease in a delay along the path will tighten or worsen the most severe hold time constraint if no parallel critical paths hide the change, and (S2) an increase in any path delay will reduce the severity of this constraint, as long as there are no parallel critical short paths.

**Proof:** (only if part) As it was for critical long paths, it is simple to show that if \( P \) is a critical short path, conditions (S1) and (S2) will be satisfied. Recall that in the early signal graphs, \(-e(f)\) is the amount of the largest hold violation. If \( e(f) \) is positive, then it represents the amount of slack in the tightest hold constraint. If \( P \) is a critical short path, then equations (3.9)-(3.12) guarantee that it determines the value of \( e(f) \), which can be calculated by simply summing the arc weights along \( P \). Increases to any delay in \( P \) will allow \( e(f) \) to increase, and reductions to any delay in \( P \) will allow \( e(f) \) to be reduced. Note that both increases and decreases to \( e(f) \) can potentially be blocked by parallel critical paths; this is due to the presence of both min and max nodes in early signal constraint graphs. A max node will prevent a change from reducing \( e(f) \) if there is another critical path through the node; similarly, a min node will block a reduction in \( e(f) \) if another critical path contains the node.
We also can show that if conditions (S1) and (S2) are satisfied, $P$ is a critical short path. Except for the qualifications for parallel paths, both conditions require that the sensitivity of the largest (or nearest) hold violation to changes in $P$ be nonzero. This implies that the event time at the sink node, $e(f)$, be determined by the sum of the arc weights along $P$, a condition which can only be true if equations (3.4)-(3.7) are satisfied for $P$. These equations imply that each arc of $P$ be a controlling arc; therefore, $P$ must be a critical short path. □

Observe that unlike the case for critical long paths, both increases and decreases in delays along a critical short path may be blocked by a parallel path. This is due to the presence of both min and max nodes in the early signal constraint graph.

The timing of a typical critical short path is illustrated in Figure 3.7-b. Examining this, we add the following remarks:

- Combining conditions (3.9)-(3.12), we see that a zero-slack critical short path constraint is of the form:

$$\sum_{i=1}^{m} (\delta_{i-1} + \delta_{i-1, i}) + q_0 - Q_m = \sum_{i=1}^{m} (\varnothing \ (L_{i-1}, L_i)) + (T_c - E'_0) - T_c + H_m$$

$$\cdots = (A(P) - 1) T_c + H_m$$  

(3.13)

Note that the terms on the leftmost side of the equation represent times required for data to propagate through logic and reach the input of the final storage device. The terms in the center and on the right represent the time available for these propagations to take place. If the path were violated, the leftmost “=” would be replaced by “<”.

- In our definition, critical short paths may contain an arbitrary number of latches. However, for most practical circuit designs, critical short paths should be free of internal latches. A circuit with a critical short path of length 2 (with one internal latch) is shown in Figure 3.9. This circuit will work correctly for the timing shown, although the minimum delay from $l_2$ to $l_3$ is by itself too short to prevent a hold violation at $l_3$. Physically, this means that whenever the circuit is started, a hold violation will occur on $l_3$ in the first cycle of operation, but then the path from $l_1$ to $l_2$ will cause the departure time from $l_2$ to increase, causing the hold violation to disappear. In some systems, these transient hold violations may be acceptable, but in others (such as restarting a stopped CPU pipeline), they are unacceptable. This property was first pointed
out by Szymanski and Shenoy [133] and implies that for a circuit to be restartable with no hold time violations, critical short paths should be no longer than one combinational segment. If we wish to verify under this condition, we simply replace the original early departure time equation with, as suggested by Szymanski [132] and Sakallah et al. [116]. This has the significant additional benefit of simplifying the early signal constraint graphs by eliminating all max nodes from the graph.

### 3.3.3 Critical Loops

Finally, a third type of critical path can limit circuit operating speeds. We call these paths critical loops, and they appear as zero- and positive-weight cycles in the late and early signal constraint graphs. A critical loop in a late signal constraint graph is shown in Figure 3.10-a; the loop structure and timing are shown in Figure 3.10-b, where $l_0 = l_{m-1}$. The formal definition of a critical loop is as follows:
Definition 3.3 A critical loop is defined to be a circular path which corresponds to a maximum-weight cycle in the late or early signal constraint graphs.

Note that this definition differs from the previous two in that it does not directly correspond to the classical definitions of Section 1.2.1.1. Classical CPM theory makes no provision for cycles in project networks; cycles are usually considered errors in planning or analysis. However, the cycles in these constraint graphs are not due to design errors, but are a natural consequence of the use of feedback in sequential circuits. The presence of critical loops was also noted in the retiming work of Ishii et. al. [66] and Lockyear and Ebeling [89]. Both groups observed that loops of transparent latches place a fundamental limit on the cycle time attainable by retiming algorithms, which relocate latches to eliminate setup time violations. Similarly, Szymanski used critical loop information to identify the minimum cycle time attainable using optimal clocking methods [132, 116, 120], which adjust clock event times to eliminate setup and hold time violations.

When a positive-weight cycle is present, some or all of the event times are undefined. Examining the constraint graph in Figure 3.10-a, such a cycle implies the following relationship:

\[ \sum_{i=1}^{m} (\Delta_{i-1} + \Delta_{i-1,i}) > \sum_{i=1}^{m} (\varnothing (L_{i-1}, L_i)) \]  

(3.14)

which states that the sum of delays in the loop is greater than the total time available for propagation around the loop. The latency of a loop is thus defined to be:

\[ \Lambda(P) = \frac{1}{T_c} \sum_{i=1}^{m} (\varnothing (L_{i-1}, L_i)) \]  

(3.15)

If the total delay of a loop exactly equals the available propagation time, then the corresponding cycle has zero weight, and all of its arrival and departure times are well-defined solutions of the equations of Table 2.2. Referring to the constraint graph of Figure 3.10-a, we see that the timing associated with a zero-weight critical loop is as follows:

\[ \forall i \in \{1,...,m\}, A_i = D_{i-1} + \Delta_{i-1,i} - \varnothing (L_{i-1}, L_i) \]  

(3.16)

\[ \forall i \in \{1,...,m\}, D_i = A_i + \Delta_i \]  

(3.17)
There is no definition for critical loops in terms of required times, slacks, and floats, as these require well-defined event times and the ability to compare these times against a local constraint (i.e. the setup or hold constraint). Unlike the long path and short path constraints, a loop constraint is not associated with any single latch, but with groups of latches arranged topologically into a loop.

Positive weight cycles can also exist in the early signal constraint graph. However, the following theorem, similar to one by Szymanski and Shenoy [133], shows that we need not check for critical loops in the early signal graph.

**Theorem 3.3.** If a critical loop constraint is satisfied for the late signal variables, it is also satisfied for the early signal variables.

**Proof:** The critical loop constraints for the early signal variables have the same form as constraints (3.16)-(3.17), but with \( a_i, d_i, \delta_i, \) and \( \delta_{ij} \) substituted for their late signal counterparts. A satisfied loop constraint has the form:

\[
\sum_{i=1}^{m} (\delta_{i-1} + \delta_{i-1},_i) \leq \sum_{i=1}^{m} (\varnothing (L_{i-1}, L_i))
\]

Since \( \delta_i \leq \Delta_i \) and \( \delta_{i,i-1} \leq \Delta_{i,i-1} \),

\[
\sum_{i=1}^{m} (\delta_{i-1} + \delta_{i-1},_i) \leq \sum_{i=1}^{m} (\Delta_{i-1} + \Delta_{i-1},_i),
\]

guaranteeing that the early signal loop constraints will be satisfied whenever the late signal loop constraints are satisfied. \( \square \)

The timing of a typical critical loop is illustrated in Figure 3.10-b. To this we add the following remarks:

- Combining conditions (3.16)-(3.17), we see that a satisfied critical loop constraint is of the form:

\[
\sum_{i=1}^{m} (\Delta_{i-1} + \Delta_{i-1},_i) = \sum_{i=1}^{m} (\varnothing (L_{i-1}, L_i)) = \Lambda(P) T_c
\]

(3.18)

Terms on the left side of the equation represent logic delays and those on the right represent the time available for signal propagations. If the constraint were violated, the leftmost “=” would be replaced by “>”.

75
• Critical loop constraints can be shown to place a lower bound on the cycle time. We define $T_{c, \text{loop}}$ for a circuit as the minimum cycle time which satisfies all critical loops. $T_{c, \text{loop}}$ can be written as:

$$T_{c, \text{loop}} = \max_{P \in \text{LOOPS}} \left( \frac{\sum_{i=1}^{m} (\Delta_{i-1} + \Delta_{i-1,i})}{\Lambda(P)} \right)$$  \hspace{1cm} (3.19)$$

where $\text{LOOPS}$ is the set of topological loops in the circuit. Note that equation (3.19) assumes that the latency of each loop is a constant. In fact, $\Lambda(P)$ will always be a constant integer for loops, as the enabling and latching event times do not appear in the loop constraint equation (3.18).

It is not necessary to enumerate the possibly-exponential number of topological loops to determine $T_{c, \text{loop}}$. As a number of researchers have observed [65, 132, 89], $T_{c, \text{loop}}$ can be calculated in polynomial time using a maximum ratio cycle algorithm to find the solution to equation (3.19). In [132], Szymanski presented a variant of Lawler’s maximum mean-weight cycle algorithm [83] which efficiently computed $T_{c, \text{loop}}$ for optimal clocking problems. Lawler’s algorithm performs a binary search over a range of possible cycle times, and each step in the search provides an additional bit of precision. Lawler’s original algorithm used the Bellman-Ford algorithm in each search step to determine whether a positive weight cycle existed, causing the worst case complexity of the inner search loop to be $O(|V||E|)$, where $|V|$ is the number of vertices in the graph, and $|E|$ is the number of edges. The algorithm is thus $O(b|V||E|)$ in the worst case, where $b$ is the number of bits of precision in $T_{c, \text{loop}}$. However, with the acceleration heuristic developed by Szymanski [132], the presence of a positive-weight cycle can be determined much more quickly. The heuristic involves periodically checking a subgraph for cycles; if a cycle is found, its weight is computed and checked. The subgraph consists of exactly one controlling arrow per node; as a result, it can be checked in $O(|V|)$ time. These extra cycle checks raise the worst-case asymptotic complexity of the algorithm to $O(b|V|^2|E|)$; in practice the enhanced algorithm runs much faster than the unmodified Lawler’s algorithm.
3.4 Identification of Critical Paths

This section describes two approaches for identifying critical paths and verifying their timing. The first implicitly checks all paths using relaxation-based solution procedures based on Algorithm SIMPLE-RELAX (Figure 3.1). Results of this procedure are used to calculate slack values, which in turn identify critical paths. The second approach uses a constructive algorithm to generate possible critical paths; if desired, we can then verify that their timing constraints are satisfied. These approaches are then evaluated and compared using the ISCAS89 sequential benchmark suite and circuits from the Michigan High-Performance Microprocessor Project.

3.4.1 Critical Path Identification by Slack Calculation

The simplest method for critical path identification relies on classical definitions of slack and float, as discussed in the introduction (Section 1.2.1.1). This section presents methods which calculate slack and float values directly from the results of a simple relaxation algorithm, such as Algorithm SIMPLE-RELAX (Figure 3.1). We begin by considering the simpler case where no critical loops are violated. We then discuss the more complicated case which arises for violated loops.

3.4.1.1 Identifying Critical Paths in the Absence of Violated Loops

If a circuit is free of violated loops, then signals at each latch will have well-defined arrival and departure times using the original timing models of Table 2.2. As a result, it is possible to calculate slack and float values with respect to these arrival times. The circuit shown in Figure 3.11 contains a single topological loop, whose timing is satisfied for the clock schedule shown. Figure 3.12 shows the late signal constraint graph for this circuit, with the corresponding actual and required event times marked by each node. The actual event times were calculated using the simple relaxation algorithm of Figure 3.1 (algorithm SIMPLE-RELAX). The setup times for all latches are zero.

Required times are obtained by recalling equations (1.2) and (1.6), but since the circuit contains cycles, the required times must also be calculated by relaxation. Using a procedure similar to SIMPLE-RELAX, required time values are computed at each node and are propagated back
through the graph until all have stabilized. Since this is again an application of the Bellman-Ford algorithm, we know that if there are no violated loops, these required times will stabilize in at most $|N|$ iterations, where $|N|$ is the number of nodes in the graph. The only thing which can prevent a solution is a positive-weight cycle in the constraint graph, and such a positive-weight cycle will only exist when a violated loop is present.

The only thing which can prevent a solution is a positive-weight cycle in the constraint graph, and such a positive-weight cycle will only exist when a violated loop is present.

Examining the late signal graph, we see that the sink has an actual event time of 1 and a required time of 0. The resulting slack at the sink is -1, and tracing back along the arrows with -1 float, we see the critical arrows marked in bold.

Examining the critical arrows, we see that there are actually two overlapping critical long paths $(l_3 \rightarrow l_1 \rightarrow l_2 \rightarrow l_5)$ and $(l_1 \rightarrow l_2 \rightarrow l_5)$ and a critical loop $(l_1 \rightarrow l_3 \rightarrow l_1)$. The loop overlaps the long paths at nodes $A_1, D_1, A_3,$ and $D_3$, and the nodes on loop have the same slack as those on the critical long path.

Only those arrows connecting nodes $D_i$ and $A_j$ correspond to actual circuit delays; all other arrows are fixed by the clock schedule and timing model. Keeping this in mind, increasing the time on any critical arrow will worsen the setup violation at latch $l_5$; reducing the time on arrows between nodes $D_1$ and $F$ will likewise reduce the amount of violation. Reductions to other times will be masked by a parallel critical path; e.g., reducing the delay on the subpath $(l_3 \rightarrow l_1)$ will not eliminate the violation due to the parallel path $(l_1 \rightarrow l_2 \rightarrow l_5)$.

Critical long paths can be enumerated using a depth-first traversal of the critical arrows, beginning at either the source or sink nodes. Beginning at the sink node, we search back along critical arrows until we reach the source; the path we obtain is a critical path. If an arrow or node is
encountered more than once during the expansion, then the expanded path $P$ contains a cycle and
the search for long paths along $P$ can be ended.

These same methods can also be used to identify critical short paths. The early signal con-
straint graph for Figure 3.11 is shown in Figure 3.13. Critical arrows are drawn in bold, and the
critical short path is $(l_1 \rightarrow l_2 \rightarrow l_4)$.

Note that neither of these slack calculations directly detected the presence of the critical
loop containing $l_1$ and $l_3$; it was instead observed as a side effect of the long path identification.

This loop prevents us from increasing the total delay $\Delta_{1,3} + \Delta_{3,1}$ or lowering the cycle time
below 10. We observe that the critical loop in Figure 3.12 consists of nodes and arcs with slack and float equal to -1, the most negative slack in the circuit.

It is easiest to identify critical loops when the corresponding cycle in the constraint graph has zero weight. When this is the case, all slacks and floats around the loop will have the same value, as shown by the following theorem:

**Theorem 3.4.** If a critical loop in a circuit has zero weight, then all of the slacks and floats around the loop will have equal values.

*Proof:* From Definition 3.3, if a critical loop has zero weight, then the maximum-weight cycle in the constraint graph will have zero weight, and thus well-defined values will exist for all the actual and required event times in the graph. Let $P_{\text{loop}}$ be the path corresponding to the critical loop. At each node in the loop, $e(v) = \max_{u \in P(v)} [e(u) + t_{u,v}]$ and $r(v) = \min_{w \in S(v)} [r(w) - t_{v,w}]$. Since $P_{\text{loop}}$ is a zero-weight cycle, any increase $de$ in a value of $e(v)$ will cause all the actual event times in the loop to increase by $de$. Similarly, a reduction $dr$ to $r(v)$ will cause all required event times in the loop to be reduced by $dr$. This implies that $e(v) = e(u) + t_{u,v}$, $u \in P(v), u \in P_{\text{loop}}$ and $r(v) = r(w) - t_{v,w}$, $w \in S(v), w \in P_{\text{loop}}$. Substituting variables and eliminating $t_{u,v}$ gives $r(u) - e(u) = r(v) - e(v)$. Comparing this with equation (1.3), we conclude that the slack of each node $v$ in the loop is equal to that of its predecessor $u$ and that all slacks in the loop are equal. Float is defined by equation (1.4) as $f(u \rightarrow v) = r(v) - e(u) - t_{u,v}$. Substituting for $e(u)$ gives $f(u \rightarrow v) = r(v) - e(v)$, and thus all floats in the loop also share the same value. \qed

Theorem 3.4 does not imply that zero-weight critical loops will have the most negative slack in a circuit, only that the slacks of the nodes in each zero-weight loop will have equal values. To find zero-weight loops we can thus perform a depth-first search over all sets of arrows and with the same float to find critical loops, but a more practical approach may be to simply search the subgraph consisting only of controlling arrows. Either case will provide a linear-time procedure for identifying one of the critical loops of a circuit (if one exists), although the total number of critical loops may be exponential in the number of latches.
3.4.1.2 Identifying Critical Paths in the Presence of Violated Loops

When the critical loop(s) in a circuit are violated, the above procedures for identifying and extracting paths break down. Since a violated loop corresponds to a positive weight cycle in the constraint graph, the arrival and departure time equations will have no solution. One way to force a solution is to use the model of Figure 3.3, which clips departure times to obtain stable values.

However, this approach also fails to provide adequate critical path information, as illustrated below. Figure 3.14 shows the late signal constraint graph for the circuit of Figure 3.11 where the delay from \( l_3 \) to \( l_1 \) (\( \Delta_{3,1} \)) has been increased to 6. This causes the critical loop containing \( l_3 \) and \( l_1 \) to be violated for the original cycle time of 10. Using the modified latch model of Figure 3.3, the event times stabilize to the values shown. However, the positive-weight cycle is again a problem when we attempt to calculate required times, as it prevents the required time relaxation from stabilizing. If we attempt to trace critical paths along the set of controlling arrows (drawn in boldface in Figure 3.14), a number of paths appear violated, but as we know from Figure 3.12, most of these apparent critical paths are unrelated to the critical loop \( l_1 \rightarrow l_3 \rightarrow l_1 \) and the critical long path \( l_3 \rightarrow l_1 \rightarrow l_2 \rightarrow l_5 \).

![Figure 3.14: Clipping Departure Times during Relaxation](image-url)
3.4.1.3 Strategies for Slack-Based Critical Path Extraction

The example of the previous section illustrates the difficulty of identifying critical paths when loops are violated. No solution exists to the original timing model; and although a solution can be found for the clipped model, we are unable to calculate required times or to unambiguously trace critical paths.

Fortunately, in most situations it is sufficient to identify only those critical paths at \( T_{c, \text{min}} \), the minimum cycle time of the circuit or at \( T_{c, \text{loop}} \), the critical loop frequency. In such an approach, \( T_{c, \text{loop}} \) can be calculated using Lawler’s algorithm, or \( T_{c, \text{min}} \) can be obtained using an optimal clocking algorithm [116, 132, 120]. Choosing one of these as the target cycle time, we can rerun the relaxation and obtain the critical long path (or paths) and any zero-weight loops. Using slack information, we can also identify nearly-critical long and short paths, but no such information is available for loops without enumerating and checking all the topological loops in the circuit. The difference is that slack and float are only defined with respect to the sink node, which does not appear in critical loops. A maximum ratio cycle algorithm can find the critical loop, but will not provide any information about the second or third (etc.) most constraining loops in the circuit.

However, timing optimization is typically done iteratively, in which analysis and optimization steps are interleaved. For any such approach, the above procedure will be adequate as long as a circuit does not contain a large number of violated loops with differing weights. Such a circuit could require an impractical number of iterations, stalling the optimization process.

An alternative approach would be to treat latches in a circuit as edge-triggered devices, much has been done in classical applications of CPM to latched circuits. However, in this case we only select a sufficient subset of latches to make the constraint graph acyclic. To decouple the arrival and departure times of a latch, we pick a required arrival time for the latch and compute the latch departure time based on that value. The latch departure times are then fixed at the calculated value. This decouples the actual latch arrival time \( A_i \) from the latch departure time \( D_i \), eliminating the \( A_i \rightarrow D_i \) arrow from the constraint graph. For each modified latch, the \( S \rightarrow D_i \) and \( A_i \rightarrow F \) arrows in the constraint graph are modified to reflect the new arrival and departure times. This is a
conservative approach, as it does not allow the departure times of the fixed latches to change during the optimization. It has the advantage that slacks and floats also reflect the loop constraints. However, the selection of latches to fix is arbitrary, as is the selection of the new required arrival time and departure time for each fixed latch. This approach is explored further in Chapter V, where we consider CPM-based algorithms for timing-driven design.

3.4.2 Critical Path Identification by Construction

In the previous section, we presented an approach for identifying and verifying critical paths based on CPM-style approaches. Actual event times were calculated in a forward relaxation through the constraint graph, and then required times were calculated in a reverse relaxation. This procedure worked well for circuits that were free of loops or for which all critical loops could be guaranteed satisfied. However, it was seen to break down when violated loops were present, since the arrival and departure times were undefined or, if the clipping modification was used, a large number of paths falsely appeared violated.

This section describes an alternate approach based on the enumeration of possible critical paths, called candidate paths, using heuristic information to reduce the number of paths enumerated. The approach is similar to the relaxation methods of Section 3.4.1 and can provide insight into their operation. It is significantly more complex, as it carries more information forward through the timing verification. However, it is also a more robust procedure for obtaining critical path information in the presence of violated critical loops. No slacks or floats are computed, so that we have less information about sub-critical paths, but when the algorithm terminates, it has identified the most constraining acyclic path(s) to each latch; i.e., the path(s) which produces the latest (or earliest) signal arrival at the latch input. Zero- and positive-weight loops are identified as a side effect of the algorithm.

3.4.2.1 Preliminaries

A path which partially satisfies the criticality conditions of Sections 3.1 or 3.2 is called a candidate path. Although candidate paths may not be critical, every critical path must also be a candidate path and any path which is not a candidate path cannot be critical. If conditions (3.4)-
(3.6) are satisfied for a path \( P \), then we call \( P \) a candidate long path. If condition (3.7) is also satisfied, \( P \) is a critical long path. If conditions (3.9)-(3.11) are satisfied for a path \( P \), then \( P \) is called a candidate short path. If condition (3.12) is also satisfied, \( P \) is a critical short path.

### 3.4.2.2 Path Extension Algorithm

Figure 3.15 sketches an algorithm that constructs the candidate long paths in a circuit. The algorithm for candidate short paths is similar. Each iteration in the algorithm generates successively longer candidate paths, with the \( i \)th iteration generating all the candidates of length \( i \). The paths generated in iteration \( i \) are constructed by extending the paths constructed in iteration \( i-1 \). The extensions are performed by identifying the latest arriving input to each latch. If the corresponding path is of length \( i-1 \), then it is extended as long as (1) the resulting extended path does not contain a cycle, and (2) the latch at the end of the path is transparent. Since they produce the latest arrival times and flow through transparent latches, the set of paths extended in algorithm EXTEND-LATE is identical to the set of candidate long paths. If parallel candidate paths produce the same arrival time at a latch, they are grouped together and are extended as a single path in successive iterations.

All candidate paths of length \( i \) can be generated from those of length \( i-1 \). This is guaranteed by the following theorem, similar to Lemma 5.2 in [132]:

**Theorem 3.5.** If a path \( P = l_0 \rightarrow l_1 \rightarrow \ldots \rightarrow l_m \) is a candidate path of length \( m \) that ends at latch \( l_m \), then \( P^* = l_0 \rightarrow l_1 \rightarrow \ldots \rightarrow l_{m-1} \) must also be a candidate path.

**Proof:** This follows directly from the critical path definitions. The constraints that are implied by \( P \) being a candidate path are a proper superset of those that imply that \( P^* \) be a candidate path.\( \square \)

A simple illustration of the algorithm EXTEND-LATE is shown in Figure 3.16. The circuit being analyzed is the circuit of Figure 3.11, but with the loop violation that complicated slack-based path identification. Each column in the table shows paths produced in the corresponding iteration. Late arrival times associated with each path are also shown, and are in the local frame-of-reference of each latch. The algorithm correctly identifies the critical long path \( l_3 \rightarrow l_1 \rightarrow l_2 \rightarrow l_5 \). In iteration 2, the violated loop involving \( l_1 \) and \( l_3 \) is detected twice, on the extension of paths to each latch.
Figure 3.17 illustrates the short path extension algorithm (Algorithm EXTEND-EARLY) for the same circuit. Early arrival times associated with each path are shown. In the first iteration, a hold violation on $l_4$ appears due to the shortness of the path $l_2 \rightarrow l_4$; however, the violation is reduced in the next iteration due to the extended path $l_1 \rightarrow l_2 \rightarrow l_4$.
3.4.2.3 Performance Analysis

The maximum number of iterations required by the algorithms EXTEND-LATE and EXTEND-EARLY is bounded by Theorem 3.6, which shows that for timing verification purposes, it is only necessary to consider simple paths. A long path or short path is simple if it contains no cycles, and a loop is simple if it contains no cycles other than the loop itself. If the simple paths are error-free, then the timing constraints of all other (composite) paths must also be satisfied.

**Theorem 3.6.** If the timing constraints of all simple long paths, short paths, and loops in a circuit are satisfied, then the timing constraints of all other paths will also be satisfied.

**Proof:** To prove this, we show that the constraints implied by a path containing an internal cycle are covered by independently verifying the cycle-free path and the separated cycle. Without loss of generality, we consider the path shown in Figure 3.18, a critical long path with an internal cycle. For simplicity, we assume that all clock and synchronizer delays are zero. For this to be a critical long path, the constraints shown in the associated list must be satisfied. However, the constraints shown will also be satisfied as long as

\[ P_1 = l_0 \rightarrow \ldots \rightarrow l_i \rightarrow \ldots \rightarrow l_j \rightarrow l_i \rightarrow \ldots \rightarrow l_m \]

is not a violated critical long path and

\[ P_2 = l_i \rightarrow \ldots \rightarrow l_j \rightarrow l_{j+1} \rightarrow \ldots \rightarrow l_k \rightarrow l_i \]

is not a violated critical loop. Constraints (0)-(j), and (l)-(m) are required in order for \( P_1 \) to be a critical long path, and constraint (z) is necessary to ensure that it is satisfied. Constraints (i+1)-(k+1) will be satisfied if \( P_2 \) is a satisfied critical loop. The remaining constraints, (k+2)-(k+j-i+1), duplicate constraints (i+2)-(j) and are also satisfied when \( P_1 \) is a satisfied critical path. The argu-
ment is similar when more than one internal cycle is present and for critical short paths and critical loops.

Theorem 3.6 implies that the longest simple candidate path will have at most \(|L|-1\) segments, where \(|L|\) is the number of latches in the circuit. Since a simple candidate path can contain no internal cycles, each latch in a circuit can occur at most once in such a path, making the longest possible simple candidate path be one that extends through every latch in the circuit. The length of such a path is \(|L|-1\).

Algorithms EXTEND-LATE and EXTEND-EARLY thus require at most \(|L|\) iterations to generate and verify all candidate paths up to length \(|L|-1\). (The additional iteration is to recognize that no more paths can be extended.) Each iteration will examine as many as the \(|E|\) edges in the circuit to identify new candidate paths. The worst case performance of this algorithm is thus \(O(|L||E|C_{\text{loop}})\), where \(C_{\text{loop}}\) is the cost of manipulating and extending paths in each iteration. A more realistic performance estimate is obtained by observing that the algorithm needs only to run for \(|P_{\max}|+1\) iterations, where \(P_{\max}\) is the longest candidate path in the circuit. The worst case

![Figure 3.18: Critical Long Path with Internal Cycle](image-url)
value of $|P_{max}|$ is $|L| - 1$; in practice it is much lower, and can usually be bounded by a small constant (less than 10).

We have yet to specify $C_{loop}$, the cost of manipulating path data in each iteration. Two operations must be performed: (1) we must extend any candidate paths and (2) we must determine whether a possible candidate is a loop. We represent paths recursively, reusing subpaths from previous iterations to construct each new candidate path. Figure 3.19 illustrates the construction of candidate paths for the example of Figure 3.16 and shows the extension of paths performed in each iteration. The dashed arrows represent loops that were identified but not extended, and the boldface arrows mark the final set of candidate paths. Since there can be at most $|E|$ parallel paths extended in each iteration, the cost of each iteration (excluding step (2) above) remains at worst $O(|E|)$ and the additional storage required in each iteration is also $O(|E|)$. If the number of fanins to each node is bounded, then the storage requirement reduces to $O(|L|)$. Thus the maximum memory utilization is either $O(|L||E|)$ or $O(|L|^2)$. Using $|P_{max}| + 1$ as a bound on the number of iterations, these reduce to $O(|P_{max}||E|)$ and $O(|P_{max}||L|)$, respectively.

In each iteration, we must also search the path structures to determine whether a path to be extended contains a specified latch. This adds a significant runtime cost to the algorithm, as each examination involves looking into a tree up to $|P_{max}|$ levels deep, with each level branching to up to $|L|$ subpaths. Since there are potentially an exponential number of parallel paths, a depth-first search to determine whether they contain a latch $l$ can add an exponential cost to each iteration. However, if we mark already-examined subpaths during the depth-first search, we can reduce the lookup cost to at worst $O(|P_{max}||E|)$, the maximum size of the path storage. If the number of parallel paths is small, the $|E|$ term reduces to a small constant, keeping the lookup cost low when $|P_{max}|$ and the number of parallel paths is small. We believe this to generally be the case, and observe it to be true for the examples presented in Section 3.5. However, the worst-case performance of the algorithm is $O(|L|^2|E|^2)$ as a result of the loop checks.

An alternative to the above approach is to represent paths as bit vectors in which each bit corresponds to a latch and is set to 1 if the latch is in the path and 0 if it is not. This allows lookups
to be performed in constant time, but each path extension involves copying $|L|$ bits. This is clearly impractical for even moderately-sized circuits and unlike the previous approach, costs are not reduced when path lengths are small. For these reasons, we chose to use the tree-based approach in our implementation and in the experiments presented in Section 3.5.

3.4.2.4 Comments and Limitations of the Path Extension Algorithms

As illustrated in Figure 3.19, the construction of candidate paths is very similar to the unrolled timing constraint graphs of Wallace and Sequin’s ATV program [139] and the computational expansions described by Ishii and Leiserson [65]. Both approaches expand sequential circuits into equivalent acyclic circuits to obtain timing constraints. The differences among approaches are in how they relate to critical loops in the constraint graph. In ATV, Wallace and Sequin limit the unrolling with a user-specified parameter and do not describe the possibility of cyclic timing constraints. Ishii and Leiserson handle these constraints implicitly by using the Bellman-Ford algorithm to check for the existence of a violated loop; they do not address the problem of identifying critical paths, although the maximum-weight cycle can be easily identified using the methods of Section 3.4.1.

In developing the path extension approach, our attempt was to identify as many violated paths as possible when the desired cycle time was below the critical loop frequency $T_{c,\text{loop}}$. However, we cannot guarantee that all violated paths will be identified, since to make the algorithm
practical, we only extend candidate paths; these are the paths that produce the latest (or earliest) arrival times at each latch. Paths that produce earlier arrival times are eliminated; although they may also contain timing violations, the magnitude of these violations will be smaller than those of the candidate paths.

The check for cycles adds a significant cost to the extension procedure. When there are violated loops in the circuit, these checks identify loops and allow us to directly compute the minimum cycle time for the loop and the delay reduction necessary to make each loop feasible. However, we observed in Figure 3.16 that critical loops can be identified more than once; in fact, the algorithm will identify a loop containing $n$ latches $n$ separate times, once at each latch in the loop.

At and above the critical loop frequency $T_{c,\text{loop}}$, the arrival and departure time calculations are equivalent to those of the relaxation algorithm SIMPLE-RELAX. At cycle times above $T_{c,\text{loop}}$, no loops will appear in the analysis; all will be broken by the clock or subsumed by another path. When the cycle time is equal to $T_{c,\text{loop}}$, the path extension approach will not propagate signal times to the next latch $l$ when a path rejoins itself to become a loop; however, since the loop corresponds to a zero-weight cycle, eliminating this propagation will not affect the timing at latch $l$. As a result, we see that Algorithm SIMPLE-RELAX can be derived from Algorithm EXTEND-LATE by simply removing the path extension and cycle checking functions. The upper bounds on the number of iterations required by both algorithms is the same, and are in both cases due to the maximum path length in the circuit. After $|L| - 1$ iterations, there are no more latches for Algorithm EXTEND-LATE to add to an existing acyclic path. Similarly, after $|L| - 1$ iterations, Algorithm SIMPLE-RELAX will have computed the timing of the longest possible critical long path, and failure of the arrival and departure times to stabilize will only be due to a violated critical loop. This same line of reasoning was used by Szymanski and Shenoy in Lemma 2.5 of [133] which proved that the arrival and departure times computed by Algorithm SIMPLE-RELAX will converge in at most $|L| - 1$ iterations, if they converge at all.
3.5 Timing Verification Experiments

To explore the different procedures for timing verification and critical path identification, we tested our implementations on circuits from the ISCAS89 sequential benchmarks and on circuits obtained from the Michigan High Performance Microprocessor project. The microprocessor circuits included the result register data path (rddpath) and the program counter datapath (pcdpath) and were two-phase level-sensitive designs. They were both of moderate size: the rddpath circuit contained 572 gates and 82 latches and pcdpath contained 4367 gates and 512 latches. Because the ISCAS89 circuits are single-phase edge-triggered circuits, we performed the transformation used by Szymanski to convert them to two-phase level-sensitive circuits [132]. The transformation is illustrated in Figure 3.20. Combinational logic blocks were duplicated and placed between alternating $\Phi_1$ and $\Phi_2$ latches and inputs and outputs were also duplicated and replaced with level-sensitive latches. The transformed ISCAS89 circuits ranged in size from 20 gates and 16 latches to over 16000 gates and 4166 latches. To indicate that “doubled” versions were used, the name of each ISCAS89 benchmark was changed to begin with a letter “d”. All circuits were clocked with a symmetric non-overlapping 2-phase clock with 50% duty cycle. Assuming zero clock skew and hold times, this eliminated the possibility of hold violations and made the maximum feasible cycle time $T_{c,\text{max}} = \infty$. A unit delay model was used for the ISCAS89 circuits, and the microprocessor circuits were analyzed using delays mapped from a commercial process [135].

![Original Circuit](image1)

![Transformed Circuit](image2)

Figure 3.20: Transformation to Obtain Two-Phase Circuits

Figure 3.21 contains a plot of the number of iterations and cpu time required to verify d15850, one of the larger benchmark circuits. As modified, d15850 contains 1396 latches and was verified at different fractions of its minimum cycle time, $T_{c,\text{min}}$. The circuit was verified using
four algorithms: the simple relaxation algorithm (SIMPLE-RELAX), the modified relaxation algorithm that clips departure time (CLIP-RELAX), the relaxation algorithm that checks for loops (ACCELERATED-RELAX), and the path extension algorithm (EXTEND-LATE). Note that all algorithms require the same number of iterations until \( T_c = 0.95T_{c,\min} \), at which point a critical loop is violated. As a result, the number of iterations required by the simple relaxation approach jumps to its maximum value \( |L| \) and the number of iterations required by the relaxation algorithm with clipping also jumps, but quickly descends as the amount of the violation increases. The number of iterations for the path extension algorithm increases steadily as more latches become transparent and path lengths increase and the relaxation algorithm with loop checks requires only a few iterations across the entire range of cycle times shown.

As could be expected, CPU times for the path extension algorithms reflect the overhead required to manipulate the extra path information that they maintain. At large values of \( T_c \), there is very little penalty, due to the shortness of the paths examined (evidenced by the small number of iterations). However, as \( T_c \) decreases and more latches become transparent, the cost of path extension increases until eventually it becomes impractical.

It was also interesting to compare the run times of the algorithms as a function of circuit size, or the number of latches in each circuit. CPU times required for the algorithms are shown in
Figure 3.22 on a log-log plot. The plot shows the result of verifying the test circuits at
\[ T_c = 0.9T_{c, \text{min}} \]. The runtimes for each of the algorithms fell roughly into straight lines, with the
slopes of the lines corresponding to the exponent \( m \) in the complexity function \( O(|L|^m) \). The
simple relaxation algorithm exhibited an approximately cubic time complexity, and the other algo-

To further observe the length of the longest path in each circuit, we plotted the lengths of
the longest paths produced by the path extension algorithm for each of the benchmark circuits at a
variety of cycle times. At \( T_c = T_{c, \text{min}} \), the longest observed paths were typically 2 to 4 segments
long, but in one circuit were as long as 13 segments. As the cycle time decreased, these path
lengths increased gradually, until at \( T_c = 0.8T_{c, \text{min}} \), the longest observed path was 17. As the
plot in Figure 3.23 shows, these path lengths were relatively uncorrelated with circuit size and
increased only slightly as the cycle time decreased. Results for the transformed ISCAS89 bench-
marks and those for the microprocessor circuits were similar.
3.6 Conclusions

We have discussed a pair of approaches for identifying critical paths that can extend through level-sensitive latches. The first, based on relaxation techniques, was simpler, faster, and more closely based on the original CPM-based approach to the identification of critical paths in combinational circuits. However, as we saw in Section 3.4.1.2, it is difficult for procedures based on this approach to produce meaningful results in the presence of violated loops, which cause the original CPM definitions to break down. To deal with this problem, we proposed a second algorithm which constructively generates paths and which can report critical long paths even when violated loops are present. The path construction complicates the verification process, as the second algorithm carries the additional overhead of updating paths in each cycle and checking for the presence of loops in the extended paths. As demonstrated in Section 3.5, in some cases this added cost is moderate. However, if the number of transparent latches in a circuit is large, or if a large number of parallel paths have the same delay, the cost of the cycle checks used by the extension algorithm will be prohibitive. Regardless, if the critical loops in a circuit are known to be satisfied or if the circuit is acyclic, the relaxation-based procedure is favored for its simplicity, speed, and the availability of slack and float information that can be used to identify sub-critical paths.
4.1 The Optimal Clocking Problem

In this chapter, we examine the application of the models of Chapter II to the problem of optimally clocking a circuit containing level-sensitive latches. Recall that the optimal clocking problem is defined as follows: given a fixed circuit structure, find a clock schedule which optimizes some aspect of circuit behavior. The most common optimal clocking problem is that of cycle time minimization, in which the clock parameters are chosen to obtain the minimum cycle time of a circuit. However, other problems could also be defined, an example of which is the safety margin maximization problem [44], which fixes the cycle time at a known feasible value and then seeks to maximize a safety margin parameter added to each setup and hold time in the circuit. The resulting clock schedule is thus the “safest” clock schedule for the cycle time given, in that it is the most tolerant of variations in clock skew, device delays, and other parameters that affect the setup and hold constraints of a circuit. In either case, the defining characteristic of an optimal clocking problem is that the only parameters treated as variables are those related to clock generation and distribution; all others are assumed fixed.

Our focus is specifically on optimal clocking problems involving level-sensitive latches, as these are especially complicated due to the ability of latches to propagate signals asynchronously when enabled. Although a number of researchers had examined the problem previously (Section 1.2.2), we start with the results of Sakallah, Mudge, and Olukotun [105, 114, 116] as they provide an elegant and complete formulation. We describe problems which arise with their formulation and simplifications which have been made to ensure correctness of results and to reduce
computation times. Among these simplifications, we describe the reduction by Szymanski [132] which greatly reduces the set of optimal clocking constraints and relate it to the critical path definitions of Chapter III. We also show that for a certain class of clock schedules, the optimization problem can be reduced to a maximum-ratio cycle problem. We then show that the general optimal clocking problem is a special case of a problem which we call min-max linear programming (mmLP). After briefly discussing the characteristics of these problems, we consider cases for which mmLP problems can be solved by a single linear program, relating these to the restrictions of Sakallah et. al. For more general cases, we present a branch-and-bound algorithm and a mixed integer linear program (MILP) formulation which can be used to obtain solutions. These methods allow us to present the first exact solutions for optimal clocking problems and to compare them to solutions obtained using existing simplifying assumptions.

4.2 The General Formulation of the Optimal Clocking Problem

Table 2.2 in Chapter II presents a set of constraints which must be satisfied to guarantee correct operation of a latch-controlled circuit. They are an extension of the constraints first presented by Sakallah et. al. to describe the operation of circuits with latches. Observing the similarity of these constraints to those in linear programs, Sakallah et. al. formulated the optimal clocking problem as a linear program by approximating the constraints of Table 2.2 with linear equations and inequalities. Min and max functions were “relaxed” and replaced with inequalities, and latching events were constrained to occur in a fixed order, eliminating the mod function from the time shift operator $\mathcal{O}(L_i, L_j)$. This original formulation was referred to as the “General System Timing Constraints” (GSTC) [116], and yielded a linear program which could be solved to find the minimum cycle time of a circuit. The solution of the LP also specified other parameters of the clock schedule, including the time of the latching edge of each clock signal and the widths of each clock phase. However, Sakallah et. al. observed that the linear program solutions they obtained did not always correspond to correct clock schedules. Verification at the clock schedules they obtained often indicated hold time violations in the circuits being studied. A number of attempts were made to postprocess the LP solutions to eliminate these hold violations, but none could reliably produce the optimum clock schedule.
4.3 Restrictions to Simplify Optimal Clocking Solutions

A number of restrictions can be made to reduce the complexity of optimal clocking problems and which eliminate the problems with the Sakallah et al.’s GSTC. This section describes several; each limits the range of possible clocking solutions in order to more easily obtain a solution. Some make restrictions on the behavior of latches or the type of clock signals used; others constrain the circuit structure.

4.3.1 The Latch Synchronization Assumption

The problems which Sakallah et al. observed with the GSTC solutions were exclusively due to violated hold time constraints. They led to the development of a restricted set of timing constraints, the “Restricted System Timing Constraints” (RSTC) [114], which were formulated to prevent hold violations a priori.

The original RSTC constraints assumed that all minimum delays were zero and that all early departure times occurred as early as possible, on the enabling edges of each latch’s clock. Using these assumptions, constraints were developed that required a minimum separation of the active intervals of the clock phases. These constraints also eliminated the early signal variables from the linear programs. It was later recognized that the early departure time assumption was sufficient and that given actual minimum delays, similar constraints could be derived that instead limited the maximum amount of overlap between active intervals of clock phases.

The early departure time assumption is that the early departure times from each latch are set by the enable event, replacing the early departure time equation (2.31) with:

\[
d_i = E'_i + q_i + \delta_{Cl} = T_c - (L_i - E_i) \mod T_c + q_i + \delta_{Cl}
\]  \hspace{1cm} (4.1)

eliminating the max function from the early signal constraints. With this restriction, the early signal constraints in Table 2.2 can be reduced to obtain a set of constraints among the events in the clock schedule. It is equivalent to assuming that in the optimal solution, all latches are synchronizing (Section 2.4.2). This is a necessary condition for a circuit to be restartable without loss of data (Section 2.4.4).
In general, for each pair of connected synchronizers \( l_i \to l_j \), we can combine equations (2.27) and (2.29) to obtain a constraint:

\[
a_j = d_i + \delta_{i,j} - \varnothing(L_i, L_j) \geq H_j + Q_j
\]  

(4.2)

Substituting (4.1) and the definition of the time shift function \( \varnothing(L_i, L_j) \),

\[
(T_c - (L_i - E_i) \mod T_c) + q_i + \delta_{C_i} + \delta_{i,j} - (T_c - (L_i - L_j) \mod T_c) \geq H_j + Q_j
\]  

(4.3)

Since we know that \( 0 \leq L_i < T_c \), we also know that \( -T_c < L_i - L_j < T_c \) and we can replace each mod function using the following substitution:

\[
(L_i - L_j) \mod T_c \rightarrow (L_i - L_j) + T_c (L_i < L_j)
\]  

(4.4)

where \((L_i < L_j) = 1\) if \(L_i < L_j\) and is zero otherwise. Substituting and simplifying (4.3) gives:

\[
L_j - E_i + T_c \left( (L_i < E_i) - (L_i < L_j) \right) \leq \delta_{C_i} + \delta_{i,j} + q_i - Q_j - H_j
\]  

(4.5)

Constraints involving the same clock events can be combined, leaving at most \(3n^2\) event interaction constraints, where \(n\) is the number of clock events.

### 4.3.2 Szymanski’s Reduction

Given the RSTC reduction, the relaxed constraints still produced large linear programs. In addition to the at most \(3n^2\) event interaction constraints, there are two departure time inequalities per latch and one arrival time inequality per latch-to-latch connection, producing as many as \(3n^2 + 2|L| + |E|\) constraints, where \(|L|\) is the number of latches and \(|E|\) is the number of latch-to-latch connections. With two variables per latch, there are \(2|L| + n\) variables in the relaxed linear program.

To reduce the number of constraints, Szymanski showed that the constraints of Sakallah et. al. could be reduced to a smaller set of relevant constraints, where each relevant constraint corresponded to the longest path between a pair of clock events with a given total time shift [132].
simplify the reduction, he used a different set of timing constraints which expressed all times in a single frame-of-reference. We can derive these constraints from those shown in Table 2.2 by mapping the data signal transition times, \( (a_i, A_i, d_i, D_i) \), and the enable event time, \( E_{i}^{o} \), back into the global reference frame. Each variable is remapped using the transformation \( t^{o} = t' - (T_{c} - L_{i}) \), illustrated in Figure 4.1. This transformation differs slightly from those used in Chapter II: it does not guarantee that \( 0 < t^{o} \leq T_{c} \). Instead, we allow \( t^{o} \) to possibly be negative in order to preserve the relative ordering among the event times.

Performing this transformation on each arrival and departure time variable in Table 2.2 produces the constraints listed in Table 4.1, where the event times \( (a_{i}^{o}, A_{i}^{o}, d_{i}^{o}, D_{i}^{o}) \) are in the global frame of reference. The time shift function is now simpler, and is \( E^{o}_{i} = E_{i} \) when \( e_{j} \geq e_{i} \), and is zero otherwise. The enable event time \( E^{o}_{i} \) is \( E^{o}_{i} = E_{i} \) when \( E_{i} \leq L_{i} \) and

\[
E^{o}_{i} = E_{i} - T_{c} \quad \text{when} \quad E_{i} > L_{i},
\]

this ensures that \( E^{o}_{i} \leq L_{i} \).

Szymanski represented the timing constraints with a graph similar to the Late Signal Constraint Graphs of Chapter II. A comparison of Szymanski’s Lower Bound Graph and the Late Signal Constraint graph is shown in Figure 4.2. Arrival and departure times were modeled with a single node; note that this complicates the modeling of separate clock- and data-to-output synchronizer delays and assumes that \( E^{o}_{i} + Q_{i} \leq L_{i} + q_{i} - S_{i} \). Nodes were added to correspond to each clock event; arcs ending on these nodes represent setup time constraints, while arcs from these nodes identify enabling and latching event times. Arcs were colored red or black; red arcs identi-
fied connections where a time shift of \(-T_c\) was necessary due to the time shift function \(\varnothing^o(L_j, L_i)\) or the definition of \(E^o_i\). Short path timing constraints were obtained using the latch synchronization assumption of Section 4.3.1, thus no equivalent early signal graph was described.

Shortest path algorithms were used to compute the most binding constraints between pairs of clock events; these were termed relevant constraints and corresponded to paths with the maximum value of \(w(P) - r(P)T_c\), where \(w(P)\) was the total weight of arcs in a path \(P\) and \(r(P)\) was the number of red arcs. These are similar to the candidate paths of Section 3.4.2 in that they are the paths in the graph that correspond to the latest signal arrival times and tightest timing constraints. A maximum ratio cycle algorithm was used to compute the maximum value of \(\frac{w(P)}{r(P)}\) over the set of cycles in the graph; this placed a lower bound on the cycle time, \(T_c\); this estimate was used to construct the set of relevant constraints.

The resulting constraints are of the form: \(e_i - e_j + rT_c \geq w\), where \(e_i\) and \(e_j\) are event times, \(r\) is the number of red arcs, and \(w\) is the total arc weight on the corresponding relevant path. By considering only relevant paths, Szymanski eliminated all of arrival and departure time vari-

<table>
<thead>
<tr>
<th>Clock Constraints</th>
<th>Time Shift Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>((R_i - F_i) \mod T_c \geq w)</td>
<td>(\varnothing^o(e_j, e_i) = \left[ \begin{array}{c} T_c (e_j \geq e_i) \ 0 (e_j &lt; e_i) \end{array} \right] )</td>
</tr>
<tr>
<td>((F_i - R_i) \mod T_c \geq w)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combinational Propagation Constraints and Synchronizer Macromodels</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Early Signals</strong></td>
</tr>
<tr>
<td>(a^o_i = \min_{j \in FI(i)} (d^o_j + \delta_{j,i} - \varnothing(L_j, L_i)))</td>
</tr>
<tr>
<td>(d^o_i \geq L_i - T_c + Q_i + H_i)</td>
</tr>
<tr>
<td><strong>latches:</strong> (d^o_i = \max(a^o_i + \delta_{D_j}E^o_i + q_i + \delta_{C_j}))</td>
</tr>
<tr>
<td><strong>flip-flops:</strong> (d^o_i = E^o_i + q_i + \delta_{C_j})</td>
</tr>
</tbody>
</table>

**Table 4.1:** Timing Model Expressed in a Global Frame-of-Reference
ables, reducing the linear program to contain only $n$ variables and a number of constraints proportional to $n^2$.

### 4.3.3 Fixed-Latency Clock Schedules

We describe a clock schedule as having fixed-latency if all of the clock event times are proportional to the cycle time, $T_c$, and as a result, the latencies defined in equations (3.3) and (3.15) of Chapter III are constant. For fixed-latency clock schedules, the optimal clocking problem becomes a single-variable optimization problem and we can compute the minimum cycle time directly from the Late Signal Constraint Graphs of Figure 2.10. Recall that these graphs were defined such that the sink node $F$ would have an event time less than or equal to zero if and only if all setup constraints in the circuit were satisfied. Assuming that hold time constraints were satisfied, the minimum cycle time was the value of $T_c$ for which all critical loops were satisfied and for
which $e(F) \leq 0$. By adding a single arc to the constraint graph, we can find $T_{c,\text{min}}$ with a maximum ratio cycle algorithm. We simply connect the sink to the source node with a zero-weight arc; if $e(F) > 0$ in the original graph, this will introduce a positive-weight cycle in the modified graph. The minimum cycle time is then determined by the cycle in the graph with the largest ratio of delay to latency. Checking the hold constraints in the corresponding early signal graph will determine whether this cycle time is feasible. If not, then no clock schedule with the specified latencies can be used, since from equation (3.13), for fixed latencies, each hold constraint places an upper bound on the cycle time:

$$T_c \leq \frac{\sum_{i=1}^{m} (\delta_{i-1} + \delta_{i-1,i}) + q_0 - Q_m - H_m}{\Lambda(P) - 1}, \Lambda(P) \neq 1 \tag{4.17}$$

If this upper bound is less than the minimum cycle time obtained above, modifications must be made to either the circuit structure or the path latencies to achieve a feasible cycle time. If $\Lambda(P) = 1$ and $\sum_{i=1}^{m} (\delta_{i-1} + \delta_{i-1,i}) + q_0 - Q_m - H_m < 0$, then the hold constraints will not be satisfied at any cycle time and, again, the circuit structure or path latencies must be modified to obtain a feasible clock schedule.

**4.3.4 Single-Phase Pipelines**

For certain restricted classes of circuits, the optimal clocking constraints can be simplified sufficiently to allow analytic solution of the constraints. One such class is that of simple pipelines, in which each stage connects to at most one predecessor (or successor) stage. This includes a group of circuits we call *circular pipelines*, in which latches are arranged in a ring [112]. This section describes a simple reduction of the general constraints to pipelines controlled by a single-phase clock, i.e. all latches have the same enabling and latching events.

Reviewing Table 2.2, the late signal constraints for a single-phase pipeline are:

$$A_i = D_{i-1} + \Delta_{i-1,i} - T_c \tag{4.18}$$

$$A_i \leq T_c - S_i + q_i \tag{4.19}$$
\[ D_i = \max(E_i' + Q_i + \Delta C_i, A_i + \Delta D_i) \]  \quad (4.20)

Note that since each pipeline stage has a single fanin, the max function is eliminated from the late arrival time equation (4.18). Since we assume single-phase operation, \( \varnothing(L_i - 1, L_i) = T_c \) and \( E_i' = E_j' \) for all latches. To further simplify the constraints, we define \( T_1 = T_c - E_i' \) and call \( T_1 \) the width of the active interval of the clock signal \( \Phi_1 \). Combining the departure time equation (4.20) with the arrival time equation (4.18) gives:

\[
A_i = \max_{k \in \{1 \ldots n\}} (T_c - T_1 + \sum_{j=1}^{k} \Delta_{i-j, i-j+1} - kT_c)
\]  \quad (4.21)

where \( n = i \) for simple pipelines and is equal to the number of stages for circular pipelines. For simplicity, clock skew and synchronizer delays are omitted and we assume that all indices are modulo \( n \). Combining this with the setup constraint (4.19) gives:

\[
\max_{k \in \{1 \ldots n\}} (T_c - T_1 + \sum_{j=1}^{k} \Delta_{i-j, i-j+1} - kT_c) \leq T_c - S_i
\]  \quad (4.22)

This places a lower bound on the cycle time:

\[
\forall i, \forall k, T_c \geq \frac{1}{k} \left( \sum_{j=1}^{k} \Delta_{i-j, i-j+1} + S_i - T_1 \right)
\]  \quad (4.23)

For circular pipelines, an additional constraint requires that signals return to their starting point no later than their original departure time:

\[
A_i = T_c - T_1 + \sum_{j=1}^{n} \Delta_{i-j, i-j+1} - nT_c \leq T_c - T_1
\]  \quad (4.24)

This reduces to:
which requires that the minimum cycle time be at least as large as the average delay through the pipeline stages.

The early signal constraints can be similarly reduced. Reviewing Table 2.2, the early signal constraints are:

\[
a_i = d_{i-1} + \delta_{i-1, i} - T_c
\]  

(4.26)

\[
a_i \geq H_i + Q_i
\]  

(4.27)

\[
d_i = \max (E_i' + q_i + \delta_{C_i} a_i + \delta_{Di})
\]  

(4.28)

Substituting the departure time equation into the arrival time equation gives:

\[
a_i = \max_{k \in \{1 \ldots n\}} (T_c - T_1 + \sum_{j=1}^{k} \delta_{i-j, i-j+1} - kT_c)
\]  

(4.29)

where \( n = i \) for simple pipelines and the number of stages for circular pipelines. Again we omit clock skew and synchronizer delays and assume indices are modulo \( n \). Combining this with the hold constraint (4.27) gives:

\[
a_i = \max_{k \in \{1 \ldots n\}} (T_c - T_1 + \sum_{j=1}^{k} \delta_{i-j, i-j+1} - kT_c) \geq H_i
\]  

(4.30)

Note that this constraint differs from (4.22) in that it is only necessary for one of the arguments to the max function to be greater than \( H_i \); in (4.22) all of the arguments must be small enough to satisfy the setup constraint. Equation (4.22) produces the following constraints on \( T_c \):

\[
\forall i \left( \exists k \neq 1, T_c \geq \frac{1}{k-1} \left( \sum_{j=1}^{k} \delta_{i-j, i-j+1} - H_i - T_1 \right) \right) \vee (T_1 \leq \delta_{i-1, i} - H_i)
\]  

(4.31)
Constraints (4.23), (4.25), and (4.31) describe the set of feasible clock schedules described by the variables \( (T_c, T_1) \). Methods for finding optimal solutions within this space were described by Sakallah et. al. [112, 113] and Chang et. al. [18]. If we assume that all latches are synchronizing (Section 2.4.2, Section 4.3.1), then (4.31) simplifies to:

\[
\forall i, T_1 \leq \delta_i - H_i
\]

which can be used to obtain a maximum value of \( T_1 \). This can then be substituted into (4.23) which with (4.25) can be used to find the minimum value of \( T_c \).

### 4.4 min-max Linear Programming

Section 4.3 described a number of simplifications to the optimal clocking problem defined by the constraints of Table 2.2. This section presents a class of problems which include the original unrestricted optimal clocking problem, discusses the complexity of these problems, and presents two methods for finding optimal solutions.

We define the problem of min-max linear programming (mmLP) as the optimization of a linear objective function subject to a set of linear, min, and max constraints. Its general form is:

\[
\text{optimize:} \quad c^T x \\
\text{subject to:} \quad A x = b, \quad x \geq 0 \\
\quad x_i = \min(C_i x + c_i), \quad i \in \text{MIN} \\
\quad x_j = \max(K_j x + k_j), \quad j \in \text{MAX}
\]

The functions \text{min} and \text{max} select the minimum and maximum entries of their vector arguments, respectively. \text{MIN} and \text{MAX} are the sets of variables which are determined by min and max functions. \( C_i \) and \( K_j \) are matrices of coefficients for min or max function \( i \); \( c_i \) and \( k_j \) are the corresponding vectors of constant terms. Although only equations have been shown, inequalities of the form “\( \geq \)” and “\( \leq \)” can be expressed by designating appropriate variables as slacks.

In addition to the optimal clocking problem, mmLP problems have arisen in a few other timing analysis applications. McMillan and Dill [98] used a simplified version of mmLP to model
and verify interface timing. Lavagno and Sangiovanni-Vincentelli [82] used linear programs with
added max constraints to pad delays and eliminate hazards in asynchronous circuits.

Despite their timing analysis applications, min-max linear programs are relatively un-
known in the operations research community. Most previously-studied problems involving min and
max constraints have a relatively simple structure and can be easily solved using linear pro-
gramming. The nearest reference to mmLP was made by Dinic [37], who described the “New Product-
New Technology” problem, an economic problem which can require mmLP solutions. As described
by Dinic, this problem is concerned with the development of new products and technologies as early
as possible, subject to the constraints that each technology cannot be established until all of the
needed prerequisite products exist, and that each product can only be produced when at least one
of some set of enabling technologies have been established. Dinic notes that this is equivalent to the
problem of calculating the time needed to propagate a rising edge through a network of AND and
OR logic gates. He also observes that both problems are naturally modeled by PERT or CPM net-
works containing both AND and OR nodes. He then presents a simple algorithm for computing the
shortest path through such a network, assuming that the delays between nodes are fixed. If instead
we allow delays to vary subject to additional cost constraints, we can formulate the problem of min-
imizing the total project time as an mmLP.

Without the added min and max constraints, mmLP problems are simply linear programs,
for which polynomial-time algorithms have been developed [49,77]. The simplex algorithm
[100,11] is a commonly-used procedure for solving linear programs. Although its worst-case com-
plexity is exponential, this complexity is almost exclusively limited to contrived cases. In practice,
the simplex algorithm exhibits polynomial runtimes and Smale showed that in the average case, the
number of pivots required by simplex grows linearly with the number of problem variables [128].
Other algorithms exist which guarantee polynomial complexity in all cases, but these often in prac-
tice run more slowly than simplex. Virtually all of these algorithms, including the simplex algo-

106

rithm, rely on the fact that the constraints in a linear program define a convex region in the solution
space. The simplex algorithm moves from vertex to vertex on the perimeter of a feasible region until
an optimum is reached. Other algorithms move through interior points as they approach the optimal
solution. The convexity of these feasible regions is the key factor allowing polynomial-time solutions of linear programming problems.

The addition of min and max constraints causes the feasible region in mmLP to become nonconvex, as illustrated in Section 4.4.1. As we shall see, at times this may not be a severe problem. For many cases of mmLP, the feasible region lies on the perimeter of a convex region. Since the optimum of a linear function will also lie on the perimeter of a convex feasible region, we can solve many mmLP problems by optimizing in the convex bounded region produced by relaxing the min and max functions. Conditions for identifying these subcases are described in Section 4.4.2. However, in other cases, the mmLP feasible region borders a nonconvex region, making it impossible to directly apply classical linear programming techniques. To handle these and other complicating cases, a branch-and-bound algorithm is presented in Section 4.4.3. Experimental results from this algorithm described in Section 4.4.4 along with results obtained using an MILP formulation and a freely-available MILP solver.

4.4.1 Characteristics of Problem mmLP

In this section we examine a number of basic characteristics of the min-max linear programming problem. We present a very simple proof of its NP-hardness and then show that it can be reformulated as a mixed integer linear program (MILP), allowing commonly-available MILP solvers to be used to find solutions. We then examine a pair of mmLP problems that illustrate important concepts used in subsequent sections.

4.4.1.1 NP-hardness

A restricted version of mmLP was proven to be NP-hard by McMillan and Dill [98] using a reduction from the problem 3-SAT. They considered the problem of optimization. Since McMillan and Dill’s problem is clearly a subset of the general problem mmLP, their proof also implies that mmLP is NP-hard.

However, for the general problem, a much simpler proof is available, based on our knowledge that the Integer Programming problem is NP-hard [49]. Specifically, we are concerned with the 0-1 Integer Programming problem (0-1 IP), which seeks an optimal objective value subject to a
set of linear constraints on a set of variables where each variable must be either zero or one. 0-1 IP is also known to be NP-hard, and it is very easy to perform the needed polynomial time reduction that converts an arbitrary 0-1 IP to an instance of mmLP.

**Theorem 4.1.** Problem mmLP is NP-hard.

*Proof:* We begin by defining an mmLP having the same set of linear constraints as the 0-1 IP. For each integer variable $x_i$, add the constraint $\max(x_i, 1 - x_i) = 1$. The only feasible solutions of this equation are $x_i = 0$ and $x_i = 1$, as shown in Figure 4.3. This completes the reduction. Note that an equivalent constraint is $\min(x_i, 1 - x_i) = 0$, which can be obtained by multiplying both sides by $-1$, adding 1, and observing the following relationship:

$$\min(x_1, x_2, \ldots, x_n) = -\max(-x_1, -x_2, \ldots, -x_n)$$  \hspace{1cm} (4.33)

Either case allows 0-1 integer variables to be represented in mmLP instances; each added min or max constraint requires one of the integer variables to have the required 0-1 integer value. Thus the general mmLP problem is NP-hard. Note that this does not prove that the general optimal clocking problem is NP-hard; the development of such a proof is an open area for research.

### 4.4.1.2 Illustrative mmLP Examples

Figure 4.4 shows an example mmLP containing a single max constraint. The feasible region is the set of darkly-shaded line segments only. Although this region is nonconvex, it bounds the shaded convex region. If we seek to minimize $x_5$, we can find an optimal solution by simply relaxing the max constraint $x_5 = \max(x_2, x_3, x_4)$, i.e. replacing it with the following set of inequalities:

![Figure 4.3: Solutions to the Equation $\max(x_i, 1 - x_i) = 1$](image-url)

108
The optimum of the relaxed problem will then be the point \((x_1 = 4, x_5 = 1)\) which coincides with the optimum of the original solution.

If we instead maximize \(x_5\), the relaxed problem is unconstrained. For the original constraints, the maximum value of \(x_5\) is 6, at the point \((x_1 = 0, x_5 = 6)\). But if we optimize using linear programming in the shaded region, there will be no solution, as the maximum of \(x_5\) is unbounded.

Another problem which can arise is illustrated with the extra constraint shown in Figure 4.4. The additional constraint is \(\frac{1}{2}x_1 + x_5 \geq 6\). The minimum value of \(x_5\) is 3 at \((x_1 = 6, x_5 = 3)\). Although it is included in the relaxed feasible region, this point is not feasible in the original mmLP. A false optimum is found because the relaxation artificially allowed the new constraint to be satisfied.

A slightly more complex mmLP is shown in Figure 4.5. Here the constraints define the non-convex region shown by the dark line. However, when the constraints are relaxed, every point in the plane becomes feasible, making the objective function unbounded.

We can find a solution by selectively relaxing constraints. In this case we relax the min constraints only and solve three subproblems, where each subproblem corresponds to one of the three terms in the max constraint being equal to \(x_5\). Each subproblem is converted into a linear program.
by replacing the max function with two inequalities and an equation. The three subproblems are illustrated on the right side of Figure 4.6; each shaded area marks the feasible region of the corresponding subproblem. After solving each subproblem, the best solution for $x_5$ is the global optimum. This decomposition of the solution space provides the framework for the branch-and-bound algorithm presented in Section 4.4.3.

Figure 4.5: More Complex min-max Linear Program

Figure 4.6: Relaxing Min and Max Constraints in the Complex mmLP

4.4.2 Single LP Solutions of mmLP Problems

As illustrated in Figure 4.4, in certain cases we can find mmLP solutions by simply solving a single linear program in which each min and max function is replaced by a set of inequalities. In this section, we describe a restricted set of mmLP problems which can always be solved with a single LP.
In general, difficulties with single LP solutions arise because we underconstrain the mmLP when we replace min and max functions with linear inequalities. The feasible region grows because the relaxation no longer enforces an important constraint: for each min and max function, one term in the function must equal the function value. A single LP solution can only be used if the optimum of the relaxed mmLP is no better than the optimum of the unrelaxed problem. Since the relaxed mmLP includes all points in the original mmLP, the relaxed optimum will be no worse than the original optimum. This implies that where a single LP solution can be used, the optimum value of the relaxed mmLP will be equal to the optimum of the original mmLP.

We begin by presenting the three conditions which we use to restrict the set of possible mmLP problems. After each is described, we present a simple proof that these conditions, if satisfied, together are sufficient to guarantee that an mmLP can be solved with a single LP. Finally, we illustrate their application on two variants of the optimal clocking problem [116].

4.4.2.1 Conditions for Single LP Solution

**Condition 1** Min and max functions should only interact with other min and max functions through homogeneous composition.

Here we define a *homogeneous composition* as a composition of min or max functions of the same type: i.e. only min functions or only max functions. Such a composition can be reduced to a single min or max function, which will always be convex. In Figure 4.5, we considered a mixed composition and saw that the resulting function (and bounded region) was non-convex. Figure 4.7 illustrates several other possible interactions of min and max functions. Of these, only the homogeneous composition (top left) is convex.

![Figure 4.7: Some Possible Interactions of Min and Max Functions](image)

**Condition 2** The optimization of the objective function and the relaxation of each min and max function must be in opposing directions.
This condition states that in order for an mmLP to be solvable by a single LP, no relaxation of a min or max function can be made in the same direction as the optimization, as this would allow relaxation to arbitrarily improve the objective function. If \( i \in \text{MIN} \), then the optimization should tend to maximize \( x_i \). Similarly, if \( i \in \text{MAX} \), the optimization should tend to minimize \( x_i \). This condition was illustrated in the example of Figure 4.4 when we attempted to maximize \( x_5 \). Relaxing the max function allowed \( x_5 \) to be arbitrarily increased and the relaxed problem was unbounded.

**Condition 3**  
Relaxation of the min and max functions must be in an opposing direction to any related linear constraints.

This condition requires that linear constraints involving min or max functions be inequalities which cannot be artificially satisfied by relaxation. This occurred in the example of Figure 4.4 when we added the additional constraint \( \frac{1}{2}x_1 + x_5 \geq 6 \). Since relaxation allowed \( x_5 \) to be increased arbitrarily, this constraint will always be satisfied in the relaxed problem and is essentially unenforceable.

### 4.4.2.2 Sufficiency of the Single LP Conditions

The following theorem shows that Conditions 1-3 are sufficient conditions to allow solution of an mmLP with a single linear program:

**Theorem 4.2.** Let \( P \) be an optimum point in a min-max linear program \( \Pi \) with feasible region \( R \), and let \( o(P) \) be the value of the objective function at point \( P \). Similarly, define \( P' \) to be the optimum point in the relaxation of \( \Pi \), \( R' \) to be the feasible region of the relaxed problem and \( o(P') \) as the value of the objective function at point \( P' \). If Conditions 1-3 are true for \( \Pi \), then \( o(P) = o(P') \) and a single linear program can be used to determine \( o(P') \).

**Proof:** We assume that Conditions 1-3 are satisfied. If Condition 1 is true, the region bounded by the min and max functions is convex. If Condition 2 is true, the optimization will push \( P' \) toward the original feasible region \( R \). The optimal point will be in \( R \) unless some min or max function must
be relaxed to satisfy a linear constraint. Since Condition 3 ensures that no such constraint will exist, 
\[ o(P) = o(P') \].

Satisfaction of Conditions 1-3 is not a necessary condition, as the variables and constraints which violate a condition may not be involved in determining the optimal mmLP solution. If Conditions 1-3 cannot be easily applied to a problem, there remains a simple approach to determine whether a single LP solution can be used: simply try it and then check the solution for an unbounded objective function or falsely satisfied constraints. If these are present, one of the methods of Section 4.4.3 should be used to solve the mmLP.

Also note that \( o(P) = o(P') \) does not imply \( P = P' \). We have observed situations where an mmLP may be solved by a single LP but for which some of the min and max functions are unsatisfied in the final solution. This occurs when the relaxed LP has multiple optimal solutions and the unsatisfied min and max functions are not critical to the optimal objective value. If exact values of these variables are desired, they can be calculated by fixing the objective function to its optimum value and solving an additional LP that seeks to minimize the outputs of relaxed max functions and maximize outputs of relaxed min functions. Alternately, the branch-and-bound algorithm or MILP formulation can be used to force all variables to their exact values.

### 4.4.2.3 Optimal Clocking Analysis

Here we apply Conditions 1-3 to the optimal clocking problem of Sakallah, Mudge, and Olukotun [116]. Examining the original constraints, shown in Table 2.2, we see that Condition 1 is violated for the early signal constraints, as the early arrival time equations
\[
a_i = \min_{j=1, n} (d_j + \delta_{j,i} - \bigodot (L_j, L_i))
\]
and early departure time equations
\[
d_i = \max (a_i + \delta_{D,i}, E_i + q_i + \delta_{C,i})
\]
can be combined to form compositions of min and max functions. We also observe that Condition 3 is violated as a result of interactions between the early departure time equations and the hold constraints \( a_i \geq H_i \). Here we require that the \( a_i \) must not be allowed to artificially increase, as this would allow relaxation to hide an actual hold violation. Relaxing the min function in \( a_i = \min_{j=1, n} (d_j + \delta_{j,i} - \bigodot (L_j, L_i)) \) is not a problem, as this only allows \( a_i \) to decrease; however, relaxing the early departure time equation
allows to increase arbitrarily. Through the interaction between \( d_i \) and \( a_i \) in the early arrival equation, \( a_i \) can now also artificially increase, allowing the relaxation to obscure possible hold violations.

If we eliminate the max function in the departure time equation, then we can eliminate both of the problems described above. This is in fact exactly what is done in the Restricted System Timing Constraints described in Section 4.3.1. \( d_i \) is assumed to take a worst-case minimum value, \( d_i = E_i' + q_i + \delta_{Ci} \), and the early signal constraints are algebraically converted into a much smaller set of constraints on pulse width and phase overlap. The only remaining nonlinear functions are the max functions in the equations \( A_i = \max_{j \in FI(i)} (D_j + \Delta_{j,i} - \varnothing (L_j L_i)) \) and \( D_i = \max(A_i + \Delta_{Di} E_i' + Q_i + \Delta_{Ci}) \). From the structure of the equations, we see that these max functions can only interact through composition, satisfying Condition 1. Considering Condition 2, we are attempting to minimize the circuit cycle time \( T_c \). Relaxing the remaining max functions can only allow \( A_i \) and \( D_i \) to increase. Since an \( A_i \) variable is never allowed to artificially decrease and since the minimum value of \( T_c \) is determined by the \( A_i \) variables (through the relation \( A_i \leq T_c + q_i - S_i \)), Condition 1 is seen to be satisfied. Considering Condition 3, we see that the setup constraints \( A_i \leq T_c + q_i - S_i \) would falsely appear satisfied if a relaxation allowed the \( A_i \) variables to decrease. Since relaxation only allows \( A_i \) variables to increase, we conclude that the setup constraints can never falsely appear satisfied and that Condition 3 is also satisfied.

**4.4.3 Methods for Exact mmLP Solution**

In this section, we present a pair of methods for exactly solving mmLP problems. The first is a simple branch-and-bound algorithm which is a straightforward elaboration of those used by McMillan and Dill [98] and Lavagno and Sangiovanni-Vincentelli [82]. The algorithm is enhanced with optimizations that significantly reduce the amount of work required when the simplex algorithm is used to find LP solutions. We also present a simple transformation that allows mmLP problems to be solved with commonly-available mixed integer linear program (MILP) solvers.
4.4.3.1 Branch-and-Bound Algorithm Structure

The algorithm is a straightforward application of branch-and-bound; the solution space may be partitioned using a tree like the one shown in Figure 4.8. The top node of the tree corresponds to the fully relaxed problem, in which each min and max function has been replaced by inequalities. Moving down the tree, each level corresponds to a min or max function, and each branch emanating a from node at that level corresponds to a term in the min or max function for which an additional constraint will be added. For a particular term \( t_i \) in a min or max function, the added constraint is \( t = t_i \), which restricts the space of possible constraints just as we did in Figure 4.6. Figure 4.8 illustrates the tree structure for the problem of Figure 4.6. The arcs explored by the algorithm are shown in bold and optimal objective values at each explored node are also shown. The depth of the tree is equal to the number of min or max functions, and the number of leaf nodes is the product of the number of arguments in each min or max function. Each leaf node defines a section of the feasible region, and the union of these sections is the nonconvex feasible region of the mmLP. If an optimum exists, at least one section will contain the optimal point.

Each internal node corresponds to a convex region which includes all of the regions defined by its descendant nodes. As a result, the optimum in this region is at least as good as the optimum in any of the leaf nodes below it. This places a bound on the solution which can be found in each subtree. In the branch-and-bound algorithm, this bound is used to eliminate subtrees from consideration whenever a comparable or better solution has already been found at a leaf outside the subtree. Also, the search of a subtree is terminated whenever a leaf node is found that provides the op-
imal solution for the subtree. Finally, if the head of a subtree does not correspond to any feasible region (due to inconsistencies among constraints), then none of the nodes below it are examined.

The branch-and-bound algorithm is implemented as a recursive procedure, which is sketched in Figure 4.9. Each node of the tree corresponds to a linear program. In certain cases, the linear program may have a special structure that will allow a faster, special-purpose algorithm to be used to find a solution [27]. If not, the simplex algorithm or some other general LP algorithm may be used.

procedure mmsolve
    generate LP for fully relaxed constraint set
    push each min/max constraint onto constraint stack
    optimal_solution = mmExpand (worst_possible_solution_value)
end

float function mmExpand (float best_so_far)
    find LP solution for current node
    if no feasible solution exists
        return worst_possible_solution_value
    else if solution is no better than best_so_far
        return best_so_far
    else if constraint stack is empty
        return solution value
    else
        pop constraint from stack
        for each slack variable $s_i$ in constraint
            add constraint $s_i = 0$ to LP
            v = mmExpand (best_so_far)
            if v is better than best_so_far
                best_so_far = v
            remove constraint $s_i = 0$ to LP
        end for
        replace constraint on stack
        return best_so_far
    end if
end

Figure 4.9: Branch-and-Bound Algorithm for mmLP Solution

4.4.3.2 General Algorithmic Improvements

In many cases, it is not necessary to solve a new LP at each node. When we are moving down from a node, the algorithm selects a constraint to convert to an equation. We can avoid solving an extra LP by choosing a constraint which trivially converts to an equation; often we can simply identify a term in the max function that already is an equation. We then make this equation explicit
and move to a lower node in the tree. In most cases, this naive optimization greatly reduces the number of LPs that must be solved.

We can also reduce the amount of computation required by only branching on a subset of the min and max constraints. The figures in Section 4.4.1.2 suggest that this is possible; and for the mmLP of Figure 4.6, solutions can be found by only branching on the single max function. This is also true for the general optimal clocking constraints; referring to the discussion of Condition 3 in Section 4.4.2, we observe that it is only necessary to branch on the max functions in the early departure time equations. Other problem-specific optimizations are also possible, including ordering of the constraints for branching, dynamically selecting a constraint during the branch-and-bound, and variable orderings based on prior knowledge about the solution space.

4.4.3.3 Algorithmic Improvements and the Simplex Algorithm

Even with the above improvements, it may still be necessary to solve a large number of linear programs to find an mmLP solution. As a result, it is important to reuse the results of existing LP solutions as much as possible. Since each move down the tree adds a single constraint to the LP to be solved, it is reasonable to surmise that the optimal solution to a child problem will be very similar to the solution of its parent LP.

Each time we solve an LP using the simplex algorithm, we must begin with a primal feasible solution, that is, a solution that satisfies all of the problem constraints but which may not be optimal. Finding this primal feasible solution requires the solution of an additional LP called an artificial program [11] (or a Phase I problem [100]); this is often the most costly part of an LP solution. In the artificial program step, additional variables, called artificial variables, are added to expand the feasible region to include a known point (typically the origin) and then the simplex algorithm is used to minimize the sum of these (nonnegative) artificial variables. If the minimum reaches zero, then the corresponding solution is primal feasible for the original problem.

We simplify the artificial program by exploiting the similarities between parent and child problems in the search tree. Each child problem differs from its parent problem in only one constraint: one of the inequalities of a relaxed min or max function in the parent is an equation for the child. Using the original problem formulation, the inequality to be modified is expressed as an equa-
tion with an added slack term, $s_i$. A feasible solution to the child problem can be obtained by simply forcing $s_i$ to zero. We can do this by solving an additional LP which minimizes $s_i$ and which begins with the optimal solution of the parent problem. This LP can be considered to be an artificial program with a single artificial variable, $s_i$ and which begins at the optimum of the parent problem. Typically, only a few simplex iterations are required to force $s_i$ to zero, much fewer than the number required by an artificial program that begins at the origin. If $s_i$ cannot be minimized to zero, then we know that the child node is infeasible. Note that this does not guarantee that the new solution is the optimum of the child problem. To obtain this we must restore the original cost function and resume the simplex iterations from the feasible point obtained in the reduced artificial program.

An alternate approach would be to use the dual simplex algorithm [100] to find the optimum of the child problem. The dual algorithm requires an initial dual feasible solution which does not satisfy the primal constraints, but whose objective value is known to be at least as good as that of the optimal LP solution. The dual algorithm iteratively moves this point towards the feasible region until a primal feasible solution is found. Since “optimality” is maintained in each iteration, the final point is guaranteed to be optimal. When the constraint $s_i = 0$ is added, the parent problem becomes primal infeasible but remains dual feasible; the dual simplex algorithm can then be used to solve the child problem in a single step.

It is not enough to merely know the values of the problem variables at the initial feasible solution; the simplex algorithm requires the set of basic variables that corresponds to that solution. For a linear program with $n$ variables and $m$ constraints, a basis vector is a vector of $m$ variables which are specified to be the only variables which can be positive in the corresponding solution. The basis matrix $B$ is an $m$ by $m$ matrix whose columns are taken from the LP constraint matrix and correspond to the constraint coefficients of the basic variables. Beginning with a feasible basis, the simplex algorithm moves to adjacent solutions by swapping non-basic variables into the basis vector one at a time, where each exchange is called a pivot step. The algorithm stops when it can no longer pivot to improve the solution.

As we move down the search tree, we seek to preserve the set of basic variables and the corresponding inverted basis matrix $B^{-1}$. The size of the basis matrix is defined by the number of
constraints. Adding constraints would enlarge the basis matrix, forcing us to recalculate $B^{-1}$. To avoid adding constraints, we simply modify the slack coefficients in the constraint matrix $A$ to convert relaxed inequalities to equations. For each inequality to be converted to an equation, the coefficient of a slack variable $s_i$ is modified to be zero. Note that this must be done after $s_i$ has been removed from the basis.

### 4.4.3.4 Transforming min-max Linear Programs to 0-1 MILPs

In an alternate approach, we can convert an mmLP to a 0-1 MILP, allowing instances of mmLP to be solved by widely-available MILP solvers (often also based on branch-and-bound approaches). Each two-variable max constraint $x_i = \max(x_j, x_k)$ is replaced with the following constraints:

\[
\begin{align*}
    x_i &\geq x_j, \quad x_i \geq x_k \\
    x_i - x_j &\leq c_i M, \quad x_i - x_k \leq (1 - c_i) M
\end{align*}
\]

where $c_i$ is a 0-1 integer variable and $M$ is a constant which must be larger than each possible difference $x_i - x_j$ and $x_i - x_k$.

The first pair of constraints ensures that $x_i$ is no less than either argument to the max function. This is the traditional approach used to replace max functions with linear constraints. When combined with these, the next two constraints ensure that either $x_i = x_j$ or $x_i = x_k$, depending on whether the 0-1 integer variable $c_i$ is 0 or 1. A max function of more than two variables can be converted by observing that any such function can be replaced by a composite of two-variable max functions, e.g. $\max(x_0, x_1, x_2) = \max(x_0, \max(x_1, x_2))$.¹

¹ Alternatively, the equality constraint can be represented using a group of 0-1 variables whose sum is unity. In this case, $\max(x_0, x_1, x_2) = c_0 x_0 + c_1 x_1 + c_2 x_2$ where $c_0 + c_1 + c_2 = 1$. However, the resulting problem is now nonlinear in the unknown variables $x_i$ and $c_i$, which may make it impossible to formulate the problem for some MILP solvers (such as [10]).
Min functions in instances of problem mmLP can similarly be converted into sets of 0-1 MILP constraints. The constraints corresponding to the equation \( x_i = \min(x_j, x_k) \) are:

\[
x_i \leq x_j, \quad x_i \leq x_k
\]

(4.37)

\[
x_i - x_j \geq -c_i M, \quad x_i - x_k \geq -(1 - c_i) M
\]

(4.38)

where again \( c_i \) is a 0-1 integer variable and \( M \) is again a large positive constant. These constraints can be justified by a discussion similar to the one above or by simply applying equation (4.33) to convert the min function to a max function and then using (4.35) and (4.36).

**4.4.4 Experimental Results**

We have implemented the branch-and-bound algorithm as discussed in Section 4.4.3. The algorithm is implemented within *splex*, a simplex-algorithm LP solver which we have developed. *splex* contains implementations of both the primal and dual simplex algorithms, but in all cases, the primal simplex algorithm (and simplified artificial program) was used to obtain solutions as the optimization moved down the search tree. We tested *splex* on a variety of mmLP problems, some of which are listed in Table 4.2. Of those shown, *minmax* is the mmLP of Figure 4.5; *Earle-setup* and *DFF-cycle* are timing macromodeling problems; *multcell* is an area-delay optimization problem, and the others are optimal clocking problems. The table lists the number of pivots and cpu seconds (on a DECstation 5000) to solve each problem for several different strategies. The columns labeled TOP list the costs of solving only the topmost LP in the solution tree. The next two columns, labeled SIMPLE, list the costs of solving the entire mmLP using the branch and bound algorithm of Section 4.4.3.1 without the enhancements of later sections. The third pair of columns (PARTIAL) contain the costs of the branch-and-bound algorithm that does not re-solve LPs at nodes that are already satisfied (Section 4.4.3.2) and the columns labeled FULL show the results of adding the enhancements of Section 4.4.3.3. The data shows that the optimizations that we developed significantly reduce the cost of branch-and-bound algorithms based on the simplex method. For the cases studied, the fully-enhanced branch-and-bound algorithm required no more than an order of magnitude more pivots than the simplex solution of the top LP in the solution tree. In one case (*Earle-setup*), more pivots
are required for the FULL results than for the PARTIAL optimization that does not reuse existing solutions. This is somewhat pathological, as the mmLP is actually solved by the topmost LP. The enhancements of Section 4.4.3.2 detect this and thus avoid re-solving the LP, but because one of the zero-valued slack variables remains in the basis, the basis-preserving algorithm (FULL) must pivot it out of the basis as it sets its coefficient to zero.

We also solved each mmLP as an MILP using the transformation of Section 4.4.3.4. MILP solutions were obtained using a modified version of lp_solve, an efficient LP solver developed at the Eindhoven University of Technology [10]. lp_solve includes both the simplex algorithm and a classical branch-and-bound IP algorithm [50]. We made two modifications to lp_solve to make it more applicable to mmLP solution. First, we parameterized the criterion used to identify integers in intermediate solutions. Second, we added an additional bound which greatly reduced the number of trial LPs explored. In the lp_solve branch-and-bound implementation, MILPs are first solved as linear programs and then the set of integer variables is checked to determine whether or not they are integers in the LP solution. If a variable $x$ has a non-integer value $x'$, two LPs are generated. One lower bounds the variable so that $x \geq \lfloor x' \rfloor$; the other places an upper bound of $x \leq \lceil x' \rceil$. In the original implementation, lp_solve always expanded both LPs, even when expanding the first led to the best possible solution for the subtree. Eliminating unnecessary second expansions reduced runtimes.
by factors up to 500 for the problems studied. For each example, the modified \textit{lp\_solve} found solutions quickly. However, \textit{lp\_solve} tends to require a larger number of pivots than \textit{splex}, suggesting that its fast running times are largely due to an optimized simplex implementation, descriptions of which are found in the LP literature [106]. We were able to solve very large optimal clocking problems with \textit{lp\_solve}, two of which are listed in Table 4.2.

We observed a strong dependence of the runtimes on the ordering of variables in the min and max functions. The optimal clocking results shown correspond to formulations in which the branch-and-bound algorithm prefers the term which is most likely to be the actual maximum; similarly, the MILP transformation assumed that the most likely maximum was selected when $c_i = 0$ in equation (4.36). In both cases the opposite formulation produced runtimes at least an order of magnitude greater.

### 4.5 Optimal Clocking Experiments

The methods of Section 4.4 now make it possible for us to solve large optimal clocking problems under the original unrestricted constraints. This allows us to address an important question: when do the unrestricted solutions differ from the more conservative RSTC optima? In an attempt to answer this experimentally, we computed the optimal clock schedules for most of the ISCAS89 benchmark circuits, where we modified each circuit to be clocked with single-phase level-sensitive latches. We assumed unit delays for each gate, and zero setup and hold times for each latch. We did not place a minimum pulse width constraint on the clock signal, allowing the degenerate solution where $R_1 = F_1$, which would correspond to edge-triggered operation. The optimal clocking solutions for each circuit are listed in Table 4.3. Three solutions are listed: the RSTC solution, the solution of the relaxed original problem (Relaxed-GSTC, the LP at the top node of the branch-and-bound tree), and the exact optimal clocking solution (GSTC).

For each of the single-phase benchmarks studied, the solutions for the restricted and unrestricted constraints were identical, suggesting that all the latches in the circuits were synchronizing at their optimal clock schedule. This reflected the fact that the minimum delays between latches were very small, typically only a few gate delays. Minimum delays in actual circuits will be an even smaller portion of the total delay: in our analysis we generously assumed that minimum
delays and maximum delays would be equal. In actual logic families, minimum delays are often as small as 25% of maximum delays [135].

Several researchers have developed procedures for padding minimum delays by inserting buffers [121, 142] or reorganizing combinational logic [78]; however, we believe that the restricted timing constraints are sufficient for most optimal clocking problems of practical interest for the following reasons:

<table>
<thead>
<tr>
<th>Circuit</th>
<th>RSTC</th>
<th>Relaxed GSTC</th>
<th>Exact GSTC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_{c, \text{min}}$</td>
<td>$F_1 - R_1$</td>
<td>$T_{c, \text{min}}$</td>
</tr>
<tr>
<td>s27</td>
<td>6</td>
<td>0-1</td>
<td>6</td>
</tr>
<tr>
<td>s298</td>
<td>10</td>
<td>1-2</td>
<td>10</td>
</tr>
<tr>
<td>s344</td>
<td>8</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>s349</td>
<td>19</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>s382</td>
<td>8</td>
<td>1-2</td>
<td>8</td>
</tr>
<tr>
<td>s386</td>
<td>11</td>
<td>0-3</td>
<td>11</td>
</tr>
<tr>
<td>s400</td>
<td>8</td>
<td>1-2</td>
<td>8</td>
</tr>
<tr>
<td>s420.1</td>
<td>12</td>
<td>1-2</td>
<td>12</td>
</tr>
<tr>
<td>s444</td>
<td>10</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>s510</td>
<td>11</td>
<td>1-2</td>
<td>11</td>
</tr>
<tr>
<td>s526</td>
<td>8</td>
<td>1</td>
<td>6.67</td>
</tr>
<tr>
<td>s526n</td>
<td>8</td>
<td>1</td>
<td>6.67</td>
</tr>
<tr>
<td>s641</td>
<td>74</td>
<td>0-6</td>
<td>74</td>
</tr>
<tr>
<td>s713</td>
<td>74</td>
<td>0-6</td>
<td>74</td>
</tr>
<tr>
<td>s832</td>
<td>10</td>
<td>0-1</td>
<td>10</td>
</tr>
<tr>
<td>s838.1</td>
<td>16</td>
<td>1-2</td>
<td>16</td>
</tr>
<tr>
<td>s953</td>
<td>16</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>s1196</td>
<td>24</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>s1238</td>
<td>22</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>s1423</td>
<td>59</td>
<td>0-2</td>
<td>59</td>
</tr>
<tr>
<td>s1488</td>
<td>16</td>
<td>1-3</td>
<td>16</td>
</tr>
<tr>
<td>s1494</td>
<td>16</td>
<td>1-3</td>
<td>16</td>
</tr>
<tr>
<td>s5378</td>
<td>24</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>s9234.1</td>
<td>56</td>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>s9234</td>
<td>56</td>
<td>2</td>
<td>38</td>
</tr>
<tr>
<td>s13207.1</td>
<td>59</td>
<td>0</td>
<td>59</td>
</tr>
<tr>
<td>s13207</td>
<td>59</td>
<td>0</td>
<td>46</td>
</tr>
<tr>
<td>s15850.1</td>
<td>82</td>
<td>0</td>
<td>82</td>
</tr>
<tr>
<td>s15850</td>
<td>82</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>s35932</td>
<td>29</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>s38584.1</td>
<td>56</td>
<td>0</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 4.3: Optimal Clocking of Single-Phase Level-Sensitive ISCAS89 Benchmarks
1. They guarantee that the circuits are restartable. (Section 2.4.4)
2. Circuits may require extensive restructuring to pad minimum delays.
3. Most processes are optimized to control the maximum delays, and minimum delays are poorly characterized, if at all.
4. Even in well-characterized processes, minimum delays are usually 30-50% of maximum delays, making it unlikely that latches will be non-synchronizing.
5. Optimal clocking problems using the restricted constraints can be solved in polynomial time, while optimal clocking problems using the general constraints are believed to be NP-hard.

We also compared optimal clocking solutions obtained with the branch-and-bound algorithm with those found using the formulas for single-phase pipelines presented in Section 4.3.4. Because of the restrictions made to obtain these constraints, they were originally only applied to simple hand-generated circuits to illustrate the non-convexity of the constraint space. Since the general branch-and-bound algorithm can successfully solve large problems exactly, a reasonable test of the technique was to try it for the simple examples of the pipeline case [113]. Solutions were identical except in a few cases where a zero-weight cycle was present in the early signal constraints, allowing the early arrival times to drift up to a solution above the minimum in order to satisfy hold constraints. Szymanski and Shenoy first observed this phenomenon [133], in which a zero-weight cycle allows the early arrival times to have any of an infinite number of degenerate solutions. The most physically meaningful solution is the lowest one, but the constraints have no way to enforce that this solution is chosen, and modifying the objective function will similarly fail to select it, as the cycle time minimization will dominate.

We believe that this problem is rare and is insignificant for all but trivial hand-constructed cases. The only way the problem can occur is for a zero-weight cycle to exist in the early signal constraint graph, and if such a cycle exists, the corresponding cycle in the late-signal graph must also have zero (or positive) weight. The only way for both cycles to have zero weight is for the minimum delays to equal the maximum delays, a condition which is probably impossible for all practical circuits. [135, 123].
4.6 Conclusions

This chapter has described a number of aspects of the optimal clocking problem. Beginning with the results of Sakallah, Mudge, and Olukotun, we discussed several simplifications which in some cases, greatly reduce the complexity of optimal clocking problems. We also presented a general formulation of a class of problems which we call min-max Linear Programming (mmLP) of which the general optimal clocking problem is an instance. With this formulation, we analyzed cases where solutions could be obtained with a single linear program and more complex cases where more sophisticated approaches were necessary. We presented two such approaches, both based on branch-and-bound algorithms, and used the branch-and-bound algorithm to obtain the first known exact solutions of general optimal clocking problems. In general, the performance of the best branch-and-bound algorithm was surprisingly good, although the solution space was heuristically sorted to cause the most likely solutions to be explored first. The performance of the branch-and-bound approach on large mmLP problems from other domains remains unexplored.

For most optimal clocking problems, the experiments of Table 4.3 suggest that the branch-and-bound approach may be unnecessary. For each of the benchmark circuits we examined, the exact optimal clocking solutions were identical to those obtained using the restricted constraints originally proposed by Sakallah et. al. While this may not be true for more highly-optimized circuits, we believe that the restricted constraints are sufficient for nearly all practical optimal clocking problems, as they produce the most robust and testable timing designs (recalling the discussion of Section 2.4.4).

Although we have discussed a variety of optimal clocking problems in this chapter, one question which we have not addressed is the typical performance improvement which can be obtained using optimal clocking techniques. This is a difficult question to answer, as it requires knowledge of “typical” circuit structures and delays. The benefits of optimal clocking depend strongly on the path length differences in a circuit; when all paths have approximately equal delays, optimal clocking algorithms are unable to “borrow” time across latches to improve performance. Alternately, when path delays differ by too wide a margin, optimal clocking analyses are unable to borrow enough time to improve performance as much as more flexible techniques which alter circuit structure or delays. Regardless, Table 5.1 of Chapter V presents a comparison of cycle
times attainable using optimal clocking techniques with those obtained assuming that all latches are edge-triggered devices. Cycle time reductions range from approximately 2% to nearly 50%; we believe this is to be a typical (though perhaps broad) range of possible performance improvements obtainable using the technique.
5.1 The Timing-Driven Design Problem

When a circuit is designed with timing constraints in mind, especially if they are fairly strenuous, we say that the design is *timing-driven*. In actual practice, this is a very broad classification, as almost all circuits are designed to achieve a certain performance target, and much of the value of the circuit is determined by its speed of operation. Section 1.2.3 described a wide variety of timing-driven design problems: logic synthesis, retiming, transistor sizing, part selection, input ordering, and placement and routing. Despite the range and variety of these problems, each timing-driven approach relied on a common framework for representing and enforcing timing constraints: the Critical Path Method.

In Chapter III we presented extended definitions of critical paths as they occur in latch-controlled circuits. In this chapter, we apply those definitions to a pair of timing driven design problems and examine the results of our extension of critical path methods to circuits with level-sensitive latches. In doing so, we hope to both demonstrate the additional optimization capability provided by the extension and to illustrate the simplicity of extending existing CPM-based optimizations to circuits with level-sensitive latches. We submit that many of the timing-driven design techniques described in Section 1.2.3 can be easily and profitably extended to better model the timing of level-sensitive latches.

We use two terms to distinguish between traditional CPM optimizations and the extension we present. *Inter-latch* optimization assumes that the arrival and departure times of all latches are fixed and thus only considers the timing properties of paths between latches. This is the traditional
approach used in the application of critical path methods to circuits with level-sensitive latches. In contrast, cross-latch optimization allows latch arrival and departure times to vary as circuit delays are changed and allows critical paths to extend through one or more transparent latches as described in Chapter III.

We define a timing-driven design problem as any circuit optimization which attempts to improve some aspect of a circuit design while maintaining or improving its timing performance. This covers a much greater range of applications than timing verification or optimal clocking, as it involves making changes to the physical structure of a circuit (and not just to the clocking methodology). The timing-driven design problems we examine are the part selection problem (Section 1.2.3.4) and the CMOS gate input ordering problem (Section 1.2.3.5). For both problems, our focus is on the late signal timings only, so we restrict our consideration to the late signal constraint graphs of Figure 2.10. For each problem, we begin with published algorithms which optimize circuits described by directed acyclic graphs. We then adapt these algorithms to the optimization of latched circuits modeled with possibly cyclic constraint graphs. The essential step in the modification is a breaking of the cycles in the constraint graphs and the addition of equivalent constraints that ensure that the loop constraints in the original graphs will be satisfied. The process of breaking cycles in constraint graphs is discussed in Section 5.2. The extended algorithms for both problems are discussed in Section 5.3 and Section 5.4. Section 5.5 concludes with a look at other problems and algorithms which can be similarly extended to circuits with level-sensitive latches.

The algorithms we describe were evaluated using ISCAS89 benchmark circuits. The original ISCAS89 circuits were synchronized using edge-triggered devices and a single-phase clock. To obtain a variety of circuit structures, we transformed these circuits into level-sensitive circuits in three ways:

1. by simply replacing edge-triggered devices with level-sensitive latches and considering the late signal constraints only (hold time constraints are ignored). These circuits have names beginning with the letter “s”, e.g., s953.
2. by replacing edge-triggered devices with pairs of level-sensitive latches controlled by alternating phases of a two-phase clock. These circuits were then retimed to minimize cycle time using a procedure similar to the one described by
Ishii [66]. These two-phase circuits have names beginning with the letter “t”, e.g., \( t953 \).

3. by using the doubling transformation described by Szymanski [132] (Chapter III). These circuits are also controlled by a two-phase clock, but have names beginning with the letter “d”, e.g. \( d953 \).

Each circuit was controlled by a symmetric clock, and all two-phase clocks were required to be non-overlapping. Since most of the algorithms we consider involve modifying circuit delays to satisfy a fixed clock schedule, this restriction is purely a convenience and does not affect the generality of the approach.

Part delays were obtained from the Texas Instruments 1-\( \mu \) CMOS standard cell library [135]. This library includes several implementations, or variants, of many simple logic functions, and each variant has different timing, area, and power dissipation properties. Unless otherwise stated, each part is assumed to be implemented using the smallest variant in the library. Also, since existing retiming algorithms do not allow for load-dependent delay models, the retimed example circuits were obtained by assuming that each gate drove 10 standard loads. Unless otherwise noted, all other analyses used actual loading effects along with standard TI pre-layout estimators for interconnect capacitance. Additional approximations for input ordering effects are detailed in Section 5.4. Specifics of each benchmark circuit used are listed in Table 5.1. \( A_{\text{min}} \) and \( A_{\text{max}} \) are the minimum and maximum areas assuming parts from the TI 1-\( \mu \) library are used and neglecting areas due to interconnect and layout irregularities. \( T_{c, \text{min-ET}} \) is the minimum cycle time of the minimum-area circuit assuming when all latches are treated as edge-triggered flip-flops and \( T_{c, \text{min-LS}} \) is the minimum cycle time of this circuit obtained using an optimal clocking algorithm.

### 5.2 Handling Cycles in Constraint Graphs

If a circuit to be analyzed contains cyclic paths clocked with level-sensitive latches, we cannot directly apply existing CPM-based techniques, since in these circuits, we must be careful not to violate the critical loop constraints. Slack information tells us by how much each delay could be increased without violating a setup constraint, but there is no similar quantity available to
ensure that the loop constraints will be satisfied. It is not difficult to construct circuits with large setup-time slacks but with a zero-weight critical loop constraint. For such circuits, increases in delays based on slack information can result in timing violations.

Table 5.1: Test Circuit Details

<table>
<thead>
<tr>
<th>circuit</th>
<th>$A_{\text{min}}$</th>
<th>$A_{\text{max}}$</th>
<th>latches</th>
<th>gates</th>
<th>$T_{c, \text{min}-ET}$</th>
<th>$T_{c, \text{min}-LS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>245.75</td>
<td>553.5</td>
<td>21</td>
<td>158</td>
<td>10.11</td>
<td>7.52</td>
</tr>
<tr>
<td>s444</td>
<td>267.25</td>
<td>628.5</td>
<td>21</td>
<td>181</td>
<td>11.13</td>
<td>7.79</td>
</tr>
<tr>
<td>s526</td>
<td>321.5</td>
<td>684</td>
<td>21</td>
<td>193</td>
<td>8.93</td>
<td>6.75</td>
</tr>
<tr>
<td>s953</td>
<td>525.5</td>
<td>1284.25</td>
<td>29</td>
<td>395</td>
<td>12.1</td>
<td>9.52</td>
</tr>
<tr>
<td>s1423</td>
<td>1029.75</td>
<td>2256.75</td>
<td>74</td>
<td>657</td>
<td>46.04</td>
<td>42.02</td>
</tr>
<tr>
<td>s9234</td>
<td>6231.5</td>
<td>16520.8</td>
<td>228</td>
<td>5597</td>
<td>35.58</td>
<td>34.89</td>
</tr>
<tr>
<td>s13207</td>
<td>9960.5</td>
<td>24797.8</td>
<td>669</td>
<td>7951</td>
<td>46.2</td>
<td>30.8</td>
</tr>
<tr>
<td>t382</td>
<td>620.75</td>
<td>928.5</td>
<td>121</td>
<td>158</td>
<td>16.01</td>
<td>12.55</td>
</tr>
<tr>
<td>t444</td>
<td>784.75</td>
<td>1146</td>
<td>159</td>
<td>158</td>
<td>16.13</td>
<td>13.31</td>
</tr>
<tr>
<td>t526</td>
<td>767.75</td>
<td>1130.25</td>
<td>140</td>
<td>158</td>
<td>13.67</td>
<td>11.75</td>
</tr>
<tr>
<td>t953</td>
<td>1020.5</td>
<td>1779.25</td>
<td>161</td>
<td>158</td>
<td>18.05</td>
<td>11.45</td>
</tr>
<tr>
<td>t1423</td>
<td>1408.5</td>
<td>2635.5</td>
<td>175</td>
<td>657</td>
<td>82.88</td>
<td>74.76</td>
</tr>
<tr>
<td>t9234</td>
<td>7585.25</td>
<td>17874.5</td>
<td>589</td>
<td>5597</td>
<td>53.22</td>
<td>26.61</td>
</tr>
<tr>
<td>t13207</td>
<td>12476.8</td>
<td>27314</td>
<td>1340</td>
<td>7951</td>
<td>87.04</td>
<td>47.17</td>
</tr>
<tr>
<td>d382</td>
<td>559</td>
<td>1174.5</td>
<td>60</td>
<td>316</td>
<td>20.22</td>
<td>10.98</td>
</tr>
<tr>
<td>d444</td>
<td>602</td>
<td>1324.5</td>
<td>60</td>
<td>362</td>
<td>22.26</td>
<td>11.6</td>
</tr>
<tr>
<td>d526</td>
<td>710.5</td>
<td>1435.5</td>
<td>60</td>
<td>386</td>
<td>17.87</td>
<td>10.7</td>
</tr>
<tr>
<td>d953</td>
<td>1343.5</td>
<td>2861</td>
<td>136</td>
<td>790</td>
<td>24.2</td>
<td>19.04</td>
</tr>
<tr>
<td>d1423</td>
<td>2224.5</td>
<td>4678.5</td>
<td>192</td>
<td>1314</td>
<td>92.08</td>
<td>59.02</td>
</tr>
<tr>
<td>d9234</td>
<td>12770.5</td>
<td>33349</td>
<td>538</td>
<td>11194</td>
<td>85.2</td>
<td>59.2</td>
</tr>
<tr>
<td>d13207</td>
<td>21061</td>
<td>56735.5</td>
<td>1642</td>
<td>15902</td>
<td>117.03</td>
<td>77.01</td>
</tr>
</tbody>
</table>

There are several alternatives for ensuring that the loop constraints are satisfied. If the optimization is formulated as a linear or nonlinear programming problem, then we can include the constraints of Table 2.2 and the loop constraints will be enforced automatically during the optimization. Unfortunately, such programs are typically complex and have large running times, making this approach impractical for large circuits. Returning to more traditional CPM-based approaches, one method for obtaining slacks is to enumerate all possible cycles in the constraint graph and calculate loop slacks based on loop delays and the available propagation time for each loop. However, this is unworkable for all but the most trivial circuits, as the number of loops in a circuit can be very large and can grow exponentially with the number of latches.

One practical approach to obtaining useful slack values is to break the cycles in the constraint graph and add simpler constraints that guarantee that loop constraints will be satisfied.
Essentially, this approach treats latches as edge-triggered devices, as has been done in classical applications of CPM to latched circuits. However, in this case we only select a sufficient subset of latches to make the constraint graph acyclic. For each such latch, we pick a required arrival time for the latch and compute the latch departure time based on that value. This decouples the actual latch arrival time $A_i$ from the latch departure time $D_i$, eliminating the $A_i \rightarrow D_i$ arc from the constraint graph. For each modified latch, the $S \rightarrow D_i$ and $A_i \rightarrow F$ arrows in the constraint graph are modified to enforce the new constraints. We thus eliminate problems due to cycles by taking a conservative approach, as we do not allow the departure times of the modified latches to change during the optimization. By modifying the $S \rightarrow D_i$ and $A_i \rightarrow F$ arrows, we cause slacks and floats to also reflect the loop constraints. The remaining acyclic graph will accurately model the timing of all unmodified latches in the circuit. After breaking loops in this manner, we can directly apply the CPM-based analysis techniques described later in this chapter and many of the others listed in Section 1.2.3.

When we break a cycle of latches, we must ensure that the timing constraints of the original graph are enforced in the ensuing analysis. This can be guaranteed with simple modifications of weights of the remaining arcs in the graph. These modifications may be done in a variety of ways, three of which are illustrated in Figure 5.1 and described below:

1. **ASAP**: Signals are assumed to depart as early (soon) as possible. Arrival time constraints are tightened to require that signals arrive in time for this early departure. The arrival time constraint becomes $A_i \leq E_i' + Q_i + \Delta_{ci} - \Delta_{Di}$ and the departure time equation simplifies to $D_i = E_i' + Q_i + \Delta_{Ci}$.

2. **ALAP**: Signals are assumed to depart as late as possible. Departure time equations are modified to reflect these later departure times. The arrival time constraint remains $A_i \leq T_c - S_i + q_i$ and the departure time equation becomes $D_i = T_c - S_i + q_i + \Delta_{Di}$.

3. **ACTUAL**: Signals are assumed to arrive and depart at the times determined by an initial timing analysis. Arrival time constraints are tightened and departure time equations are modified to reflect these times. The arrival time constraint becomes $A_i \leq T_c - S_i + q_i + \Delta_{Di}$.

131
equations are modified to enforce this constraint. The arrival and departure time constraints become $A_i \leq A'_i$ and $D_i = A'_i$, where $A'_i$ is the fixed maximum arrival time at latch $i$.

Each of the three cycle-breaking methods adds additional constraints to the circuit timing and has a different effect on the resulting CPM analysis and optimization. All three, however, fix some of the latch departure times to constant values, essentially treating a subset of latches as edge-triggered devices.

For the late-signal analysis, breaking loops using the ALAP constraints will result in a pessimistic analysis, since we are assuming that no time can be borrowed to reduce the departure time from the modified latch. Similarly, ASAP forces signals to depart as early as possible, and may constrain signals to arrive unnecessarily early. For both cases, the minimum cycle time calculated for the original graph may not be valid when the loops in the graph are broken.
The third method, ACTUAL, uses times determined by a timing analysis done prior to the optimization. The timing analysis computes arrival and departure times for each latch. For each feedback latch to be modified, signal arrival times are constrained to be no later than the arrival time as calculated, and departure times are set to the calculated departure time. Because we have not changed the arrival and departure times of these latches, the original minimum cycle time will remain feasible when cycles in the graph are broken. The limitation of this approach is that it requires a feasible timing solution at the target cycle time to obtain arrival and departure time values. From Chapter III, we know that no solution will exist if the original constraint graph contains a positive-weight cycle at the desired cycle time.

It is also of interest to consider which arcs in the graph should be removed to make the graph acyclic. Our goal should be to minimize the effects of breaking the loops on the timing of the circuit being optimized. Since each removed arc adds extra timing constraints, it would be natural to seek to break as few arcs as possible to make the circuit acyclic. This goal reduces to the problem FEEDBACK ARC SET [49], which asks “Given a directed graph \( G = (V, A) \) and a positive integer \( K \leq |A| \), is there a subset \( A' \subseteq A \) with \( |A'| \leq K \) such that \( A' \) contains at least one arc from every directed cycle in \( G \)?” Unfortunately, FEEDBACK-ARC-SET is widely known to be NP-complete; in fact it was among the first problems proven NP-complete [75].

Although the problem of finding a minimal (or boundedly small) set of arcs to break is NP-complete, we can easily find a set of arcs whose removal will be sufficient to make the graph acyclic. Figure 5.2 shows a simple loop-breaking algorithm based on a depth-first traversal of the late-signal constraint graph. The algorithm recursively traverses the graph, marking nodes as they are visited. When a mark is found, a cycle has been located and can be observed in the stack of nodes that the algorithm maintains. Algorithm BREAK-CYCLES then looks back into this stack to identify an \( A_i \rightarrow D_i \) arc, which it then removes to break the cycle. Removing this arc could have the side effect of breaking other cycles in the graph. To ensure that the graph markings accurately reflect this, after removing arc \( A_i \rightarrow D_i \), the traversal backs up to node \( A_i \) before continuing.

Algorithm BREAK-CYCLES allows for a variety of different criteria for selecting the \( A_i \rightarrow D_i \) arc to remove. The simplest criterion is to simply break the first arc found on the stack;
this is a fast and greedy heuristic. Another criterion is to choose the arc corresponding to the latch
with the latest departure time. This could be useful when the ALAP strategy is used to break loops,
but requires that an initial feasible solution exists at the target cycle time. Flow-based criteria
could be used to attempt to minimize the number of arcs which are removed, however, it is not
clear that this will have the smallest impact on circuit timing. When using the ALAP strategy to

Figure 5.2: Algorithm SIMPLE-CYCLE-BREAK
break loops, a better criterion may be to break each loop at the latch which precedes the smallest latch-to-latch delay in the loop. This will minimize the impact of fixing the latch departure time at its latest value and does not require that an initial feasible solution exist at the target cycle time.

Another very simple approach to cycle-breaking exists when circuits are controlled by an \( n \)-phase clock, where \( n > 1 \). If the latches in a circuit are arranged so that latches controlled by clock phase \( p \) fan out to latches controlled by phase \( p + 1 \) (modulo \( n \)), then we can remove all cycles from the constraint graph by selecting one of the \( n \) phases and removing the \( A_i \rightarrow D_i \) arcs for all latches controlled by the chosen phase.

### 5.3 Timing-Driven Part Selection

In this section, we examine the problem of selecting part implementations from a library to minimize area or power costs subject to timing constraints. The problem we consider is as follows: we are given a netlist of parts and a library containing discrete sets of implementations of each part, where each implementation has different drive capability, input load, and cost characteristics (area or power). We seek an implementation for each part to minimize the chosen cost characteristic subject to a fixed constraint on circuit timing, typically a minimum cycle time.

The objective of our study is to compare the results obtained by simple inter-latch optimization with the results of more complex cross-latch optimizations using the models of the previous sections. As a result, after describing some background information, we will describe implementations of several algorithms we used, some complex and some very simple. Each algorithm is implemented around the expanded late signal constraint graphs of Section 2.3. Depending on the presence of \( A_i \rightarrow D_i \) arcs in the constraint graph, we can use each algorithm to perform either inter-latch or cross-latch optimization, although it will be necessary remove some arcs to break loops in cyclic constraint graphs. In addition to examining the differences between inter-latch and cross-latch optimization, we will also briefly consider the effects of different loop-breaking strategies.
5.3.1 Background

The cell timing model we use is the model used in the majority of other transistor sizing (Section 1.2.3.3) and part selection (Section 1.2.3.4) work and is illustrated in Figure 5.3. The delay $\Delta_i$ through a part (logic gate) $G_i$ is modeled with the following equation:

$$\Delta_i = R_{\text{out}, i} \left( C_{\text{out}, i} + C_{\text{wire}, i} + \sum_{j \in FO(i)} C_{\text{in}, j} \right) = \tau_i + R_{\text{out}, i} \left( C_{\text{wire}, i} + \sum_{j \in FO(i)} C_{\text{in}, j} \right)$$ (5)

The output resistance $R_{\text{out}, i}$ is inversely proportional to the active area of the part, and the input and output capacitances $C_{\text{in}, i}$ and $C_{\text{out}, i}$ increase approximately linearly with cell size, making the intrinsic delay $\tau_i = R_{\text{out}, i} C_{\text{out}, i}$ relatively insensitive to the gate area. $FO(i)$ is the set of indices of the fanout parts of $G_i$. $\sum_{j \in FO(i)} C_{\text{in}, j}$ is the total load capacitance of all fanout parts of $G_i$. $C_{\text{wire}, i}$ is the parasitic capacitance associated with the interconnect attached to the output of $G_i$ and does not vary with the sizes of fanout devices. In the TI 1-μ library, $C_{\text{wire}, i}$ is estimated as a function of the total area of parts in the circuit and the number of fanouts of $G_i$.

![Figure 5.3: Cell Timing Model ([87])](image)

In most cases, $\sum_{j \in FO(i)} C_{\text{in}, j}$ is significant compared to $C_{\text{wire}, i}$, making the delay of a series of parts a nonlinear function of the part sizes. Increasing the size of gate $G_i$ reduces $R_{\text{out}, i}$ and the corresponding delay $\Delta_i$, but causes $C_{\text{in}, i}$ to increase, which increases the delays of parts...
fanning into \( G_i \). But although the delay model of equation (5) is nonlinear, it is known to be a convex function [43] which can be optimized using standard convex optimization techniques, some of which were listed in Section 1.2.3.3.

It is also possible to formulate the part selection problem as a linear program, where the delay vs. area (or power) constraints are approximated with linear inequalities and these inequalities are then used to formulate a linear program [10]. Solutions of the linear program can then be heuristically mapped back into the discrete set of part sizes. One advantage of this approach is that it can be easily extended to include the optimal clocking constraints of Chapter IV, allowing simultaneous solution of both optimal clocking and transistor sizing problems. Solutions obtained in this way will be able to fully exploit the cycle time borrowing property of latches to minimize cost while preserving timing correctness. A similar approach was described by Chuang et. al. [25] who combined a linearized delay model with Fishburn’s constraints for clock skew optimization [44]. They then combine these with heuristics to select from the integer solutions near the optimal LP solution. However, the complexity of the linear programming step adds a significant computational cost that is prohibitive for large circuits. For the circuits described, running times were one to two orders of magnitude larger for the LP-based approach than for a simpler critical path-based approach [87].

For large circuits, simpler approaches may be preferable which sacrifice optimality for simplicity and speed of execution. The simplicity of many critical path-based algorithms make them practical for very large circuits, on the order of several thousand gates or more, and perhaps as many as the millions of gates appearing in forthcoming designs. These CPM-based algorithms are usually quite fast, and some simple algorithms run in linear time with respect to circuit size.

In the following sections, we describe four CPM-based part selection algorithms, each of which can be easily extended to the optimization of circuits with level-sensitive latches. The first two algorithms are based on classical ideas of constraint graph compression and decompression and assume that variations in part input capacitance are negligible compared to the total load capacitance at each gate output. The third algorithm is an adaptation of the TILOS algorithm [43] and fully accounts for variable input loads. The fourth algorithm also accounts for these variations and is based on Hinsberger and Kolla’s dynamic programming algorithm for optimally sizing
chains of parts [57,58]. Although all four algorithms require acyclic constraint networks, the procedures of Section 5.2 may be used to make cyclic networks acyclic.

5.3.2 Algorithm SIZE-UP

In classical critical path methods, the most common method for optimizing the duration of a project is by compression of the constraint graph [3]. Beginning with the least expensive project specification, job times are iteratively shortened to reduce the total project time until a target completion time is met or until continued improvement is impossible (or too expensive). Jobs to be crashed (or sped up) are selected from jobs on the critical path, and are the jobs whose time can be reduced the most at the least additional cost. After the delay of a job is reduced, the constraint graph is updated to reflect new job times and a new path may be critical.

The first and simplest algorithm we consider is based on constraint graph compression and assumes that the fixed load capacitance on the output of each gate is large compared to the variations caused by changes to the input capacitances of its fanout parts. This corresponds to a situation where interconnect delays and parasitics dominate circuit timing, an assumption which may become more valid as circuit feature sizes shrink. Neglecting these variations in input capacitance, part delays decrease monotonically as part sizes increase, and the fastest version of a circuit will be the one which uses the largest parts.

Algorithm SIZE-UP is sketched in Figure 5.4. The circuit to be optimized is assumed to initially use the smallest and slowest variants of each part, and then parts are iteratively replaced with larger, faster variants until the timing specification is satisfied. Rather than attempting to evaluate the cost of resizing each part, Algorithm SIZE-UP simply assumes that any part with negative slack should be sized up to its next largest variant.

Each iteration processes nodes in reverse topological order, which preserves the actual event times and allows us to recompute required times as we move back through the network. Recomputation of required times ensures up-to-date slack information that reflects all changes made to the circuit. Parts are resized upward monotonically, causing part delays to decrease monotonically and slack values to monotonically increase. In each iteration, all parts with negative slack are resized to their next largest variants. Since slack values monotonically increase, any part which
is resized in iteration $k$ must also have been resized in all iterations $j < k$. As a result, Algorithm SIZE-UP requires at most $V_{\text{max}} - 1$ iterations, where $V_{\text{max}}$ is the maximum number of variants of any part used in the circuit. After $V_{\text{max}} - 1$ iterations, all parts with negative slack will be at their maximum sizes and further sizing will be impossible.

Computation of the actual event times requires $O(|\mathcal{N}|)$ time and the inner loop requires $|\mathcal{N}|$ iterations, where $|\mathcal{N}|$ is the number of nodes in the constraint graph. Since all operations inside the loop require constant time, the algorithm has a worst-case time complexity of approximately $O(V_{\text{max}} |\mathcal{N}|)$. Since $|\mathcal{N}|$ is linearly related to circuit size, Algorithm SIZE-UP runs in linear time with respect to circuit size.

```
generate acyclic constraint graph
repeat
    compute actual times for all parts
    set required time for sink node to zero
    for each node in reverse topological order
        compute required time for node
        compute node slack
        if (node slack < 0)
            use next largest part variant
            adjust arc weights in constraint graph
        end if
    end for
until no more changes
```

Figure 5.4: Algorithm SIZE-UP

As described, algorithm SIZE-UP works from the sink node back through the constraint network. The algorithm could also be formulated to work forward, updating actual event times as delays in the graph are modified.

### 5.3.3 Algorithm SIZE-DOWN

A second approach to the optimization of CPM project networks is to begin with all jobs planned to complete as quickly as possible and to then iteratively increase the times of jobs on non-critical paths to reduce cost. This approach has been called constraint graph decompression [3] and has two advantages: (1) the initial completion time represents the earliest possible completion of the project, and (2) the slack and float information in the network represent the exact amount of time which can be added to each job before the job becomes critical.
Independent float, $f_I$, (see Section 1.2.1.1) is associated with each arrow and represents the portion of the arrow’s float which is not affected by changes in other parts of the circuit. As a result, it is the first float which can be exploited to slow down as many gates as possible. Each delay increase will have no effect on the timing of other gates or the circuit cycle time. Unfortunately, independent float is usually only a small part of the total float in a network. The remaining float, $f_T - f_I$, can be termed *interfering float*, as it represents float which, if used, will reduce the float of other jobs in the network. Allocation of this float is a complex problem and is crucial for obtaining a good solution.

Just as Algorithm SIZE-UP was based on constraint graph compression, we define Algorithm SIZE-DOWN which performs a very simple decompression of the constraint graph. Illustrated in Figure 5.4, Algorithm SIZE-DOWN begins with the fastest implementation of each part, where again we assume that input loading variations are negligible. The minimum cycle time of the circuit can be thus easily determined by selecting the largest variant of each part and using an optimal clocking algorithm (Chapter IV). If this cycle time is greater than the desired target, then other optimizations are necessary to achieve the target cycle time.

Beginning with a feasible target cycle time, Algorithm SIZE-DOWN calculates actual times for each node in the graph. It then makes a backward pass through the network, reducing the size of all parts for which the added delay is less than the part slack. Like Algorithm SIZE-UP, Algorithm SIZE-DOWN makes no attempt to evaluate the area improvement associated with each substitution; it is instead designed to be as simple and as fast as possible. Because we neglect input load variations, part delays will increase monotonically and slack values will monotonically decrease. As a result, Algorithm SIZE-DOWN requires at most $V_{\text{max}} - 1$ iterations, where $V_{\text{max}}$ is the maximum number of variants of any part used in the circuit. The inner loop contains two passes that each require $O(|N|)$ time, giving Algorithm SIZE-DOWN a worst-case time complexity of approximately $O(V_{\text{max}} |N|)$, which again is linear with respect to circuit size.
5.3.4 Algorithm SIZE-TILOS

If we must consider the effects of variable input loads, the sizing problem becomes significantly more difficult. While larger parts have larger driving capabilities, they also have larger input loads which slow down the propagation of their input signals. As a result, the fastest circuit implementation is no longer necessarily the one using the largest parts.

One of the most widely known delay optimizers which accounts for this property is the TILOS (TImed LOgic Synthesizer) program, developed by Fishburn and Dunlop at AT&T Bell Laboratories [43]. TILOS combines a CPM-based timing analyzer with a simple quadratic programming method. An iterative procedure, TILOS first identifies the critical path or paths in a circuit and then selects one transistor from the path(s) to be resized. The transistor chosen is the one with the largest sensitivity value, which is defined as the amount of delay reduction per incremental increase in area. Although later work showed that the TILOS algorithm did not always produce optimal sizings [124], is generally seen to produce good results in a fairly short amount of time. As described by Fishburn and Dunlop, TILOS was originally developed to size individual transistors and allowed for a nearly-continuous range of sizes (subject to the increment size). However, a similar approach can be used for the part selection problem, and Lin et. al. developed a comparable procedure which also used sensitivity information to guide sizing [87].

The version of the TILOS algorithm that we use (Algorithm SIZE-TILOS) is shown in Figure 5.6. It is similar to Algorithm SIZE-UP in that it compresses a constraint graph, but uses
sensitivity information to guide the selection of parts to be resized and includes input capacitance effects in its models of delay.

The sensitivity that we use is defined as the change in delay due to a proposed resizing per unit change in area. A negative sign is included so that decreases in delay correspond to positive sensitivities. The exact definition is as follows:

\[
\rho_i = -\frac{1}{\Delta \text{Area}} \left[ \Delta \tau_i + \Delta R_{\text{out},i} C_{\text{load},i} + \min_{k \in F\text{I}(i)} (R_{\text{out},k} \Delta C_{\text{in},i}) \right] \tag{6}
\]

where \( C_{\text{load},i} = C_{\text{wire},i} + \sum_{j \in F\text{O}(i)} C_{\text{in},j} \). The first two terms in the sum correspond to changes in the drive capability of \( G_i \); the third term is a worst-case assumption about the change in part input capacitance. It assumes that the input that is least sensitive to changes in \( \Delta C_{\text{in},i} \) is also the one responsible for the latest possible arrival at an input of \( G_i \).

In each pass, actual and required times are computed for each node. The nodes sharing the smallest slack value are examined, and the node with the largest sensitivity is selected to be resized. The number of iterations required by this algorithm varies with the number of parts that must be resized, but in the worst case is \((V_{\text{max}} - 1) |M|\), where each part must be sized up the maximum number of times. Algorithm SIZE-TILOS thus has a worst-case complexity of approximately \( O((V_{\text{max}} - 1) |M|^2) \).

Figure 5.6: Algorithm SIZE-TILOS

Sizings obtained using Algorithm SIZE-TILOS are presented in Section 5.3.6. Although it is an effective tool for timing-driven area optimization, Algorithm SIZE-TILOS is based on a heu-
ristic algorithm for continuous sizing of individual transistors. It has been suggested that better algorithms exist for the discrete part-selection problem [57, 58]. The next section presents an algorithm based on a procedure which finds optimal sizings for sections of a circuit.

5.3.5 Algorithm SIZE-HINSBERGER-KOLLA

All of the algorithms presented thus far relied on heuristic assumptions to approximate the optimal sizing of a circuit. This section describes a procedure built around an algorithm for optimally sizing a chain of parts developed by Hinsberger and Kolla, who also showed that the general problem of timing driven sizing (with variable input capacitance effects) is NP-complete [57, 58].

Without loss of generality, Hinsberger and Kolla’s algorithm assumes that each part in a chain of $n$ parts can be constructed using one of $V$ variants. A possible sizing is then defined by a vector $u = (u_1, \ldots, u_n) \in \{0, \ldots, V - 1\}^n$. Each variant has an associated integer representing its area overhead $\alpha_i(u)$, which is defined to be zero for the smallest part size, so that $\alpha_i(0) = 0$, and which increases monotonically with part size. Each part also has an input load associated with it determined by its size. The algorithm can be used for any delay model which is determined solely by the part size and the sizes of its fanout parts. The core of the algorithm is a procedure which calculates the delay value $\tau(i, a, y)$ which corresponds to the smallest delay to part $i$ which can be obtained with a maximum area overhead of $a$ and an output load indicated by the index $y$. $\tau(i, a, y)$ is calculated using the following equation:

$$
\tau(i, a, y) = \min_{x, \alpha(x) \leq a} \tau(\text{pred}(i), a - \alpha_i(x), x) + \delta(x, y, \text{pred}(i), i)
$$

(7)

where $\tau(0, a, y) = 0$ for all $a$ and $y$. Using the algorithm in Figure 5.7, all the $\tau(i, a, y)$ for a circuit can be calculated in $O(n \cdot V^2 \cdot A_{\text{max}})$ time. $A_{\text{max}}$ is the maximum allowable area overhead and can be a specified parameter or determined by the sum of the maximum part overheads

$$
A_{\text{max}} = \sum_{i=1}^{n} \alpha_i(V - 1)
$$

Once the $\tau(i, a, y)$ have been calculated, the optimal sizing for a given area $a$ can be traced back through their values in $O(n \cdot v)$ time.
A similar algorithm was also developed for fanout-free trees [58], but as we have already mentioned, the sizing problem for general circuits is strongly NP-complete. To optimize arbitrary circuits, Hinsberger and Kolla proposed using their sizing algorithms to iteratively resize the most critical path or tree in a circuit. The iteration stops when either the target cycle time is obtained or it is no longer possible to reduce the delay of the most critical circuit path. Experiments in [57] suggested that the path-based approach was superior to the tree-based sizing, providing solutions of comparable quality in significantly less time. Because of these results, we make our comparisons using the algorithm of Figure 5.8 (Algorithm SIZE-HINSBERGER-KOLLA), which iteratively resizes chains of parts.

\begin{verbatim}
for a = 0 to Amax
  \( \tau(0, a, y) = 0 \) for all \( y \)
for i = 1 to n
  for y = 0 to V-1
    compute \( \tau(i, a, y) \) from formula
  end for
end for

Figure 5.7: Algorithm COMPUTE-\( \tau \) ([57])
\end{verbatim}

It is difficult to estimate the time complexity of Algorithm SIZE-HINSBERGER-KOLLA, since the number of iterations of the outer loop can potentially be very large. The number of paths in a circuit can be exponentially large; it could conceivably be necessary to resize all such paths. Worse yet, each sizing affects the timing of intersecting paths, making it possible for the algorithm to get stuck in a loop, alternately sizing and resizing two such interacting paths. These situations seem unlikely in practice, and in our experiments (Section 5.3.6), the algorithm terminates after a relatively small number of iterations. Within the iteration, the worst-case step is the call to Algo-
algorithm COMPUTE-τ with complexity $O(n \cdot V^2 \cdot A_{max})$. If we allow $A_{max} = \sum_{i=1}^{n} \alpha_i (V - 1)$, then this is potentially $O(n \cdot V^2 \cdot V^n)$. Again, in practice, much faster behavior is observed.

### 5.3.6 Experimental Results

This section describes experiments conducted using the algorithms of Sections 5.3.2-5.3.5 on the transformed ISCAS89 benchmark circuits described in Section 5.1. The experiments were designed to compare the potential of cross-latch optimization with simpler inter-latch optimizations that do not propagate timing information across latches.

Each of the algorithms we described can be used to explore the relationship between the area and the minimum cycle time of a circuit. Each circuit is capable of operating at a variety of speeds, depending on the sizing of the individual parts in the circuit. Assuming all parts are at their minimum size, we can compute a minimum cycle time for the circuit. In many cases it is possible to reduce this minimum by adding area to the circuit in the form of larger part variants. As a result, we expect an inverse relationship between area and minimum cycle time.

Figure 5.9-a shows area vs. minimum cycle time relationships for the benchmark circuit t953.bench obtained using Algorithm SIZE-TILOS. Four curves are shown, each of which corresponds to a different strategy for constructing the late-signal constraint graph used in the optimization:

1. SIMPLE: This corresponds to the simple inter-latch optimization which is the traditional CPM formulation for latched circuits. Departure times at all latches are fixed and latches are thus essentially treated as edge-triggered devices. This is the curve that would be found using the TILOS algorithm as presented in the literature [43].
2. ALAP: A full late-signal constraint graph is generated, and cycles are broken by replacing selected arcs and using the ALAP method. Arcs to be removed are arbitrarily chosen as the first arcs found to be in a cycle.
3. ACTUAL1: Again, a late-signal constraint graph is generated, but in this case, cycles are broken using the ACTUAL method, and are broken on the latch with
the latest departure time in the loop. Recall, however, that a feasible timing solution is required to obtain the arrival and departure times used in the arc replacement. For ACTUAL1, the initial timing solution was the minimum cycle time of the original (minimum-size) circuit.

4. ACTUAL2: As with ACTUAL1, a late-signal constraint graph is generated and cycles are broken using the ACTUAL method. This time, the initial timing solution was obtained by first sizing the circuit using the SIMPLE strategy to reduce latch-to-latch delays as much as possible. The arrival and departure times used to break loops were then computed at the minimum cycle time of the sized circuit.

Examining Figure 5.9-a, we see that the actual variation in area due to each optimization is fairly small; even the most aggressive timing optimizations increase circuit area by a few percent. Figure 5.9-b shows these percent increases in area for each of the four strategies. Figure 5.9-c shows the CPU time\(^1\) required by each approach at each target cycle time. These running times are seen to be directly related to the amount of additional area required; optimizations requiring larger amounts of additional area require a proportionately larger number of iterations of the TILOS algorithm.

The horizontal segments of the SIMPLE curves are due to the fact that the inter-latch approach simplifies the timing behavior of latches, essentially treating them as edge-triggered devices. The peak of each curve (where it becomes horizontal) corresponds to the minimal cycle time that can be obtained while treating the latches as edge-triggered devices. For the optimized circuits, lower cycle times may be feasible, and the lowest such cycle time can be determined using an optimal clocking algorithm (Chapter IV). The horizontal line segment thus represents the range of times which can be reached by optimally clocking the fastest implementation attainable using the SIMPLE strategy. However, the SIMPLE inter-latch optimization technique is unable to make area-delay trade-offs at these lower cycle times.

The ALAP curves correspond to an extended analysis which allows arrival times to propagate through latches during the analysis. Because of the additional flexibility of trading time across latches, the area-delay curve for this approach is lower and to the left of the SIMPLE curve, indi-

\(^1\) All CPU times in Chapter V are CPU seconds on a lightly loaded DECstation 5000/120.
Figure 5.9: Optimization of t953 using Algorithm SIZE-TILOS
cating that lower cycle times can be obtained using a comparable area or that lower areas can be achieved at comparable cycle times. Because less additional area is required, the running times for these optimizations are also less than those for the SIMPLE strategy. The best cycle time at which this approach can be used is approximately 11.2, although optimal clocking results show a minimum cycle time of the fully optimized circuit to be slightly less, as shown by the horizontal line segment. This difference is due to the extra restrictions that were added to break cycles in the constraint graph.

The ACTUAL1 curve is interesting because it is a simple horizontal line, ranging from the largest cycle time considered down to the minimum cycle time of the minimum-sized circuit. By accurately modelling the behavior of level-sensitive latches, we see that the original circuit could be feasibly clocked with cycle times as low as 11.5; this means that any cycle time above this value could be reached with no additional area. However, at lower cycle times, a critical loop is violated and no arrival and departure times are available. Since the ACTUAL1 curve shows no addition of area, it is omitted from Figure 5.9-b.

Finally, the ACTUAL2 curves show the results of optimizations that broke cycles using the ACTUAL method and times determined by an analysis of the fastest circuit attainable using the SIMPLE strategy. This provided a good initial point from which to set more appropriate arrival and departure times, and the area curves show that the resulting areas are significantly less. Surprisingly, the ACTUAL2 strategy was able to optimize at cycle times below the minimum obtained by the SIMPLE strategy. Since the ACTUAL2 strategy requires an initial timing solution to fix the arrival and departure times when removing arcs, we would expect it to be unable to optimize below cycle times of 11.8, the minimum cycle time of the initial circuit. However, since all we need are arrival and departure times, we can actually optimize at any frequency for which these values exist, regardless of whether setup violations are present. Using the algorithms of Chapter III, we can determine the minimum loop frequency to be approximately 8.7, well below the point at which Algorithm SIZE-TILOS fails to find a feasible sizing. As shown in Figure 5.9, the ACTUAL2 strategy is able to optimize circuits down to cycle times of approximately 9.4, which optimal clocking results indicate to be the minimum cycle time of the maximally-optimized circuit. Interestingly, though, the ACTUAL2 strategy is unable to find minimal areas at large cycle times.
times, probably because the optimized starting point produced an arrival time constraint that could not be satisfied by the minimum-area circuit.

Similar curves are shown in Figure 5.10 for the same circuit using Algorithm SIZE-HINSBERGER-KOLLA, based on the part selection algorithm of Hinsberger and Kolla. The same four strategies were used to generate the curves shown, and the three plots show the total area, percentage increase, and CPU time of the optimization routine. The most striking feature of Figure 5.10 is its similarity to Figure 5.9, which suggests that these two optimization algorithms find almost equally good solutions and that the solutions they find may be close to the true optima, as determined by the circuit structure, library properties, and timing constraints. We observe that SIZE-HINSBERGER-KOLLA is slightly faster, perhaps because it optimizes an entire path at a time, but the run time difference shown is almost inconsequential. Similarly, the quality of solutions differs slightly, with SIZE-TILOS finding slightly smaller implementations for a given area, but this difference is also small. However, both algorithms show similar improvements when cross-latch optimizations are used, and both produce better results when “initialized” with the timing of an optimally sized circuit from SIMPLE.

Figure 5.11 shows the fraction of latches in the example circuit which are transparent at various cycle times resulting from each of the four strategies illustrated and for both Algorithm SIZE-TILOS and SIZE-HINSBERGER-KOLLA. At large cycle times, approximately 65% of the latches are transparent; this is a side effect of the retiming algorithm and the way we model inputs of the retimed benchmarks. Input signals are assumed to arrive on the falling edge of the $\Phi_2$ clock; these inputs then propagate through combinational logic to $\Phi_1$ latches. Since we assume a non-overlapping two-phase clock with 50% duty cycles, these $\Phi_1$ will be open when these input signals arrive, causing them to be transparent even at arbitrarily large cycle times. The retiming caused an imbalance in the number of $\Phi_1$ and $\Phi_2$ latches; after the retiming, nearly 80% of the latches in t953 are controlled by the $\Phi_1$ clock. Many of these which are connected to circuit inputs are thus transparent. As the cycle time is reduced, more latches become transparent, regardless of the optimization strategy used. The largest percentage of transparent latches appears when the ACTUAL2 strategy is used, and reaches approximately 90%. We also notice that the SIMPLE
Figure 5.10: Optimization of t953 using Algorithm SIZE-HINSBERGER-KOLLA
strategy tends to cause the fewest latches to become transparent; this is probably because it is the only strategy which does not attempt to optimize across transparent latches and thus would tend not to cause latches to become transparent. However, at lower cycle times, more transparent latches appear as arrival times occur later in the clock cycle. So more transparent latches appear even though the SIMPLE strategy does not optimize across these latches.

<table>
<thead>
<tr>
<th>Cycle Time</th>
<th>Transparent Latches (fraction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>0.9</td>
</tr>
<tr>
<td>10</td>
<td>0.85</td>
</tr>
<tr>
<td>11</td>
<td>0.8</td>
</tr>
<tr>
<td>12</td>
<td>0.75</td>
</tr>
<tr>
<td>13</td>
<td>0.7</td>
</tr>
<tr>
<td>14</td>
<td>0.65</td>
</tr>
<tr>
<td>15</td>
<td>0.6</td>
</tr>
<tr>
<td>16</td>
<td>0.55</td>
</tr>
<tr>
<td>17</td>
<td>0.5</td>
</tr>
<tr>
<td>18</td>
<td>0.4</td>
</tr>
<tr>
<td>19</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Figure 5.11: Fraction of Latches in t953 which are Transparent after the Optimization

There is one subtle difference between the initial solutions used by the ACTUAL2 strategy for Algorithms SIZE-HINSBERGER-KOLLA and SIZE-TILOS. When Algorithm SIZE-HINSBERGER-KOLLA is used to optimize the area of a path, it uses a target path delay to find the smallest implementation that will satisfy the timing constraints. However, it can also be used to

Figure 5.11: Fraction of Latches in t953 which are Transparent after the Optimization
find the minimum delay of a path, regardless of total area. In this case, each path delay is reduced as much as possible during the optimization. To obtain the fastest possible initial solutions, this was the approach taken when generating the starting points for the ACTUAL2 strategy and Algorithm SIZE-HK. In most cases, this approach produced faster circuits than attainable using SIZE-TILOS, which only optimized a single part at a time. However, in a few circuits, this aggressive path optimization had side effects on neighboring paths and produced slower solutions than those found using SIZE-TILOS. These effects can be seen in a comparison of Table 5.3 and Table 5.5, as presented in the discussion below.

The next experiments we describe explore points in the area-delay curves for a variety of circuits using inter-latch and cross-latch optimizations. For each of the algorithms of Sections 5.3.2-5.3.5, we compare the results of cross-latch with simpler inter-latch optimizations. Optimizations are made at two target cycle times:

• $T_H$: the minimum cycle time attainable while treating latches as edge-triggered devices throughout the timing analysis. For Algorithm SIZE-TILOS and t953.bench, this is the point in Figure 5.9 where the SIMPLE curve becomes horizontal, at $T_c = 12.67$.

• $T_L$: the minimum cycle time attainable using inter-latch optimization but treating latches as level-sensitive devices during the optimal clocking calculation. In Figure 5.9, this is the leftmost point of the SIMPLE curve, at $T_c = 11.82$.

Both cycle times are obtained by first optimizing the latch-to-latch paths as much as possible; assuming that $\emptyset(L_i, L_j) = cT_c$, where $c$ is a constant fraction over all pairs of connected latches $i$ and $j$, the best possible cycle time for edge-triggered circuits (or using inter-latch optimization) can be obtained by maximizing the minimum slack in the constraint graph, regardless of the value of $T_c$. $T_H$ is the minimum cycle time when latches are treated as edge-triggered devices in the optimal clocking analysis; for SIZE-TILOS it is also the lowest cycle time at which inter-latch optimization can be used to manage area-cycle time trade-offs. $T_L$ is the minimum cycle time obtained using an optimal clocking algorithm which fully models latch behavior. This is the absolute lowest cycle time which can be reached and represents the “best effort” of the sizing algorithm.
to reduce circuit delays. These cycle times differ for each circuit and also for each algorithm used, reflecting the optimization capabilities of each algorithm, since each algorithm uses a different procedure to optimally size latch-to-latch paths.

Table 5.2 describes the experiments using Algorithm SIZE-TILOS at $T_H$. The first column of numbers lists $T_H$, the fastest cycle time attainable using edge-triggered techniques. The column labeled $\Delta A_{\text{SIMPLE}}$ lists the percentage increase in area necessary to reach $T_H$ using inter-latch optimization; $CPU_{\text{SIMPLE}}$ contains the running times of the optimizations. The next pair of columns, $\Delta A_{\text{ALAP}}$ and $CPU_{\text{ALAP}}$, list the percentage increase and running times of the cross-latch optimization, where again the ALAP method was used to break the first latch identified in each loop. The final pair of columns contain the ratio of added area and running times of each cross-latch optimization with respect to its corresponding simpler inter-latch optimization. From the table, $\Delta A_{\text{SIMPLE}}$ ranged from almost zero to over 15%, and $\Delta A_{\text{ALAP}}$ ranged from near zero up to 5.57%. On average, the cross latch optimization required only 44% as much additional area as the inter-latch optimization, and results were obtained in 62% of the time.

The results at $T_L$ are similar and are listed in Table 5.3. For these experiments, the columns labelled $\Delta A_{\text{ACT2}}$ and $CPU_{\text{ACT2}}$ represent the increase in area and the corresponding running time when the ACTUAL2 strategy is used to break cycles. The relative improvement is less since we are comparing the cross-latch optimization with the same initial circuit obtained with the inter-latch optimization of Table 5.2. However, the cross-latch optimization still requires an average of 47% less additional area than the simpler inter-latch approach and uses 75% of the running time.

The data in Table 5.4 and Table 5.5 describe experiments which compare inter-latch and cross-latch optimizations using Algorithm SIZE-HINSBERGER-KOLLA. Algorithm SIZE-HINSBERGER-KOLLA optimizes a path at a time, making it necessary to compute a target delay for each critical path to be sized. Since the best cycle time for each circuit was initially unknown, the target delay for each critical path was set to $-\infty$ in order to reduce each path delay as much as possible. In some cases this would cause the path to be over-optimized, resulting in the use of
unnecessarily large parts on the path, which in turn caused the delays of side paths to be unnecessarily large. As a result, the best edge-triggered cycle times, $T_H$, were at times be greater than those obtained using SIZE-TILOS, which performed a more gradual resizing. However, the corresponding level-sensitive cycle times, $T_L$, were often quite a bit lower, as they benefitted from the additional path delay reductions.

Table 5.2 shows the results of optimizations using SIZE-HINSBERGER-KOLLA at $T_H$.

To provide a more fair comparison, the inter-latch optimization (SIMPLE) at $T_H$ was rerun with $T_H$ as the target cycle time; this eliminated the over-optimization that had been initially performed. However, because inter-latch optimization techniques cannot be used at cycle times below $T_H$, the areas for $T_L$ (Table 5.5) are those of the original optimization. Comparing these with
cross-latch optimizations, we see that at $T_H$, the cross-latch approach requires an average of 42% as much additional area and uses an average of 56% as much CPU time. As with Algorithm SIZE-TILOS, the ALAP method was used to break loops on the first latch identified in each loop. The ACTUAL2 strategy was used at $T_L$; results of this are shown in Table 5.5. Again, less additional area was required (44% of the extra area required by the inter-latch optimization) and slightly less CPU time was used (98%).

Although Algorithms SIZE-TILOS and SIZE-HINSBERGER-KOLLA are more appropriate algorithms to use for general part selection problems, in Section 5.3.2 and Section 5.3.3 we presented the algorithms SIZE-UP and SIZE-DOWN which used a simplified model of part delays and more naive optimization algorithms. Although simple, these algorithms are of interest to us because they also provide examples of critical-path based optimizations which we can use to

<table>
<thead>
<tr>
<th>circuit</th>
<th>$T_L$</th>
<th>$\Delta A_{\text{SIMPLE}}$</th>
<th>$CPU_{\text{SIMPLE}}$</th>
<th>$\Delta A_{\text{ACT2}}$</th>
<th>$CPU_{\text{ACT2}}$</th>
<th>$\Delta A_{\text{ACT2}}/\Delta A_{\text{SIMPLE}}$</th>
<th>$CPU_{\text{ACT2}}/CPU_{\text{SIMPLE}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>6.179</td>
<td>8.65%</td>
<td>0.378</td>
<td>4.98%</td>
<td>0.308</td>
<td>0.58</td>
<td>0.81</td>
</tr>
<tr>
<td>s444</td>
<td>6.531</td>
<td>7.58%</td>
<td>0.437</td>
<td>6.45%</td>
<td>0.496</td>
<td>0.85</td>
<td>1.14</td>
</tr>
<tr>
<td>s526</td>
<td>6.462</td>
<td>3.27%</td>
<td>0.363</td>
<td>3.27%</td>
<td>0.512</td>
<td>1.00</td>
<td>1.41</td>
</tr>
<tr>
<td>s953</td>
<td>7.347</td>
<td>15.37%</td>
<td>5.55</td>
<td>11.85%</td>
<td>5.01</td>
<td>0.77</td>
<td>0.90</td>
</tr>
<tr>
<td>s1423</td>
<td>36.316</td>
<td>3.74%</td>
<td>5.67</td>
<td>2.35%</td>
<td>5.12</td>
<td>0.63</td>
<td>0.90</td>
</tr>
<tr>
<td>s9234</td>
<td>21.63</td>
<td>6.94%</td>
<td>539.2</td>
<td>6.99%</td>
<td>578.24</td>
<td>1.01</td>
<td>1.07</td>
</tr>
<tr>
<td>s13207</td>
<td>26.752</td>
<td>2.15%</td>
<td>336.5</td>
<td>1.05%</td>
<td>275.9</td>
<td>0.49</td>
<td>0.82</td>
</tr>
<tr>
<td>t382</td>
<td>9.5471</td>
<td>2.13%</td>
<td>0.734</td>
<td>0.60%</td>
<td>0.437</td>
<td>0.28</td>
<td>0.60</td>
</tr>
<tr>
<td>t444</td>
<td>8.845</td>
<td>2.90%</td>
<td>4.5</td>
<td>0.83%</td>
<td>0.796</td>
<td>0.29</td>
<td>0.18</td>
</tr>
<tr>
<td>t526</td>
<td>9.933</td>
<td>0.78%</td>
<td>0.652</td>
<td>0.52%</td>
<td>0.679</td>
<td>0.67</td>
<td>1.04</td>
</tr>
<tr>
<td>t953</td>
<td>11.815</td>
<td>6.52%</td>
<td>5.84</td>
<td>0.88%</td>
<td>2.05</td>
<td>0.14</td>
<td>0.35</td>
</tr>
<tr>
<td>t1423</td>
<td>64.32</td>
<td>2.33%</td>
<td>5.93</td>
<td>1.51%</td>
<td>6.13</td>
<td>0.65</td>
<td>1.03</td>
</tr>
<tr>
<td>t9234</td>
<td>25.4686</td>
<td>6.16%</td>
<td>654.2</td>
<td>5.60%</td>
<td>695.1</td>
<td>0.91</td>
<td>1.06</td>
</tr>
<tr>
<td>t13207</td>
<td>37.8636</td>
<td>2.08%</td>
<td>498.9</td>
<td>0.69%</td>
<td>240.9</td>
<td>0.33</td>
<td>0.48</td>
</tr>
<tr>
<td>d382</td>
<td>9.283</td>
<td>7.60%</td>
<td>2.19</td>
<td>3.89%</td>
<td>1.3</td>
<td>0.51</td>
<td>0.59</td>
</tr>
<tr>
<td>d444</td>
<td>10.316</td>
<td>6.06%</td>
<td>2.23</td>
<td>3.36%</td>
<td>1.62</td>
<td>0.55</td>
<td>0.73</td>
</tr>
<tr>
<td>d526</td>
<td>10.624</td>
<td>2.96%</td>
<td>1.7</td>
<td>0.99%</td>
<td>1.05</td>
<td>0.33</td>
<td>0.62</td>
</tr>
<tr>
<td>d953</td>
<td>15.73</td>
<td>12.32%</td>
<td>7.76</td>
<td>2.08%</td>
<td>6.34</td>
<td>0.17</td>
<td>0.82</td>
</tr>
<tr>
<td>d1423</td>
<td>53.74</td>
<td>3.46%</td>
<td>22.7</td>
<td>1.78%</td>
<td>19.1</td>
<td>0.51</td>
<td>0.84</td>
</tr>
<tr>
<td>d9234</td>
<td>48.818</td>
<td>9.93%</td>
<td>&gt;2100</td>
<td>3.96%</td>
<td>&gt;2100</td>
<td>0.40</td>
<td>N/A</td>
</tr>
<tr>
<td>d13207</td>
<td>61.465</td>
<td>3.37%</td>
<td>&gt;2100</td>
<td>2.95%</td>
<td>&gt;2100</td>
<td>0.87</td>
<td>N/A</td>
</tr>
<tr>
<td>averages</td>
<td>5.38%</td>
<td>2.88%</td>
<td>0.53</td>
<td>0.75</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: SIZE-TILOS results at $T_L$
explore the effects of cross-latch vs. inter-latch optimizations on timing-driven design algorithms.

Table 5.6 and Table 5.7 contain results for Algorithm SIZE-UP that mirror those in Table 5.4 and Table 5.5. Similar results for Algorithm SIZE-DOWN are in Table 5.8 and Table 5.7. For both algorithms, \( T_H \) and \( T_L \) were obtained for each circuit by assuming that the largest variant of each part was used. At \( T_H \), inter-latch optimization could be used to trade slack in non-critical parts for reductions in circuit area, but at \( T_L \), no inter-latch optimization was possible. Times and areas of corresponding cross-latch optimizations are shown, as are area and execution time reductions.

Again, the amount of additional circuit area is seen to be reduced by an average of approximately 50%. Because no SIMPLE area optimization was possible at \( T_L \), \( \Delta A_{\text{SIMPLE}} \) in Table 5.7 and

<table>
<thead>
<tr>
<th>circuit</th>
<th>( T_H )</th>
<th>( \Delta A_{\text{SIMPLE}} )</th>
<th>( CPU_{\text{SIMPLE}} )</th>
<th>( \Delta A_{\text{ALAP}} )</th>
<th>( CPU_{\text{ALAP}} )</th>
<th>( \frac{\Delta A_{\text{ALAP}}}{\Delta A_{\text{SIMPLE}}} )</th>
<th>( CPU_{\text{ALAP}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>6.64</td>
<td>8.34%</td>
<td>0.484</td>
<td>2.64%</td>
<td>0.277</td>
<td>0.32</td>
<td>0.57</td>
</tr>
<tr>
<td>s444</td>
<td>7.67</td>
<td>5.52%</td>
<td>0.562</td>
<td>2.15%</td>
<td>0.277</td>
<td>0.39</td>
<td>0.49</td>
</tr>
<tr>
<td>s526</td>
<td>6.73</td>
<td>6.07%</td>
<td>0.539</td>
<td>1.01%</td>
<td>0.187</td>
<td>0.17</td>
<td>0.35</td>
</tr>
<tr>
<td>s953</td>
<td>8.58</td>
<td>8.61%</td>
<td>2.21</td>
<td>3.28%</td>
<td>1.37</td>
<td>0.38</td>
<td>0.62</td>
</tr>
<tr>
<td>s1423</td>
<td>35.4</td>
<td>8.01%</td>
<td>12.8</td>
<td>4.61%</td>
<td>2.80</td>
<td>0.58</td>
<td>0.22</td>
</tr>
<tr>
<td>s9234</td>
<td>26.6</td>
<td>9.65%</td>
<td>225.6</td>
<td>5.08%</td>
<td>152.5</td>
<td>0.53</td>
<td>0.68</td>
</tr>
<tr>
<td>s13207</td>
<td>28.1</td>
<td>5.24%</td>
<td>227.2</td>
<td>3.25%</td>
<td>202.6</td>
<td>0.62</td>
<td>0.89</td>
</tr>
<tr>
<td>t382</td>
<td>9.68</td>
<td>2.13%</td>
<td>0.461</td>
<td>0.97%</td>
<td>0.359</td>
<td>0.45</td>
<td>0.78</td>
</tr>
<tr>
<td>t444</td>
<td>11.2</td>
<td>1.18%</td>
<td>0.523</td>
<td>0.29%</td>
<td>0.297</td>
<td>0.24</td>
<td>0.57</td>
</tr>
<tr>
<td>t526</td>
<td>10.5</td>
<td>0.68%</td>
<td>0.476</td>
<td>0.16%</td>
<td>0.253</td>
<td>0.24</td>
<td>0.53</td>
</tr>
<tr>
<td>t953</td>
<td>14.0</td>
<td>1.59%</td>
<td>1.44</td>
<td>0.37%</td>
<td>0.844</td>
<td>0.23</td>
<td>0.58</td>
</tr>
<tr>
<td>t1423</td>
<td>62.8</td>
<td>5.04%</td>
<td>10.75</td>
<td>2.91%</td>
<td>2.47</td>
<td>0.58</td>
<td>0.23</td>
</tr>
<tr>
<td>t9234</td>
<td>34.9</td>
<td>9.54%</td>
<td>144.2</td>
<td>4.62%</td>
<td>101.1</td>
<td>0.48</td>
<td>0.70</td>
</tr>
<tr>
<td>t13207</td>
<td>51.8</td>
<td>4.09%</td>
<td>145</td>
<td>2.92%</td>
<td>145.9</td>
<td>0.71</td>
<td>1.01</td>
</tr>
<tr>
<td>d382</td>
<td>13.3</td>
<td>7.25%</td>
<td>1.43</td>
<td>2.24%</td>
<td>0.804</td>
<td>0.31</td>
<td>0.56</td>
</tr>
<tr>
<td>d444</td>
<td>15.3</td>
<td>4.90%</td>
<td>1.58</td>
<td>1.83%</td>
<td>0.797</td>
<td>0.37</td>
<td>0.50</td>
</tr>
<tr>
<td>d526</td>
<td>13.5</td>
<td>5.49%</td>
<td>1.74</td>
<td>0.84%</td>
<td>0.758</td>
<td>0.15</td>
<td>0.44</td>
</tr>
<tr>
<td>d953</td>
<td>17.2</td>
<td>6.74%</td>
<td>6.85</td>
<td>2.46%</td>
<td>4.39</td>
<td>0.36</td>
<td>0.64</td>
</tr>
<tr>
<td>d1423</td>
<td>70.7</td>
<td>7.42%</td>
<td>31.9</td>
<td>3.43%</td>
<td>6.20</td>
<td>0.46</td>
<td>0.19</td>
</tr>
<tr>
<td>d9234</td>
<td>55.6</td>
<td>13.63%</td>
<td>838.9</td>
<td>8.17%</td>
<td>504.6</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>d13207</td>
<td>47.8</td>
<td>0.65%</td>
<td>112.2</td>
<td>0.12%</td>
<td>42.5</td>
<td>0.19</td>
<td>0.38</td>
</tr>
<tr>
<td>averages</td>
<td>5.66%</td>
<td></td>
<td>0.42</td>
<td>2.64%</td>
<td></td>
<td>0.56</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: SIZE-HINSBERGER-KOLLA results at \( T_H \)
Table 5.7 represents the area increase due to using the largest part variants and no CPU time comparisons were made.

For Tables 5.5-5.9, Algorithms SIZE-UP and SIZE-DOWN used a simplified version of the TI 1-µ library which assumed that each part output drove 10 standard loads, so that results shown in Tables 5.5-5.9 cannot be directly compared with previous results. However, we can make several interesting observations from these tables alone. The first observation we make is that Algorithm SIZE-DOWN produced consistently smaller (better) solutions than Algorithm SIZE-UP. This is probably because Algorithm SIZE-DOWN was slightly more discriminating in its selection of parts to resize. Recall that in each pass, Algorithm SIZE-DOWN reduces the size of each part with slack greater than the resulting increase in delay; this tends to minimize slacks to small nonnegative values. In contrast, Algorithm SIZE-UP increases the sizes of parts until all slack values are greater than or equal to zero. It shows no preference to sizings which produce

<table>
<thead>
<tr>
<th>circuit</th>
<th>$T_L$</th>
<th>$\Delta A_{\text{SIMPLE}}$</th>
<th>$CPU_{\text{SIMPLE}}$</th>
<th>$\Delta A_{\text{ACT2}}$</th>
<th>$CPU_{\text{ACT2}}$</th>
<th>$\frac{\Delta A_{\text{ACT2}}}{\Delta A_{\text{SIMPLE}}}$</th>
<th>$CPU_{\text{ACT2}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>4.86</td>
<td>27.4%</td>
<td>0.816</td>
<td>12.7%</td>
<td>0.812</td>
<td>0.46</td>
<td>1.00</td>
</tr>
<tr>
<td>s444</td>
<td>5.70</td>
<td>20.9%</td>
<td>1.06</td>
<td>9.82%</td>
<td>1.33</td>
<td>0.47</td>
<td>1.25</td>
</tr>
<tr>
<td>s526</td>
<td>6.59</td>
<td>12.8%</td>
<td>0.742</td>
<td>4.20%</td>
<td>0.453</td>
<td>0.33</td>
<td>0.61</td>
</tr>
<tr>
<td>s953</td>
<td>7.62</td>
<td>36.4%</td>
<td>4.15</td>
<td>9.04%</td>
<td>2.78</td>
<td>0.25</td>
<td>0.67</td>
</tr>
<tr>
<td>s1423</td>
<td>34.3</td>
<td>9.13%</td>
<td>13.6</td>
<td>5.88%</td>
<td>9.60</td>
<td>0.64</td>
<td>0.70</td>
</tr>
<tr>
<td>s9234</td>
<td>20.2</td>
<td>16.0%</td>
<td>316.5</td>
<td>9.98%</td>
<td>257.5</td>
<td>0.63</td>
<td>0.81</td>
</tr>
<tr>
<td>s13207</td>
<td>23.3</td>
<td>9.08%</td>
<td>414</td>
<td>4.35%</td>
<td>165.2</td>
<td>0.48</td>
<td>0.40</td>
</tr>
<tr>
<td>t382</td>
<td>5.47</td>
<td>5.36%</td>
<td>0.566</td>
<td>1.09%</td>
<td>0.570</td>
<td>0.20</td>
<td>1.01</td>
</tr>
<tr>
<td>t444</td>
<td>5.87</td>
<td>7.01%</td>
<td>0.918</td>
<td>3.15%</td>
<td>1.32</td>
<td>0.45</td>
<td>1.44</td>
</tr>
<tr>
<td>t526</td>
<td>6.51</td>
<td>4.07%</td>
<td>0.555</td>
<td>2.64%</td>
<td>1.28</td>
<td>0.65</td>
<td>2.31</td>
</tr>
<tr>
<td>t953</td>
<td>12.0</td>
<td>14.4%</td>
<td>2.56</td>
<td>0.71%</td>
<td>0.972</td>
<td>0.05</td>
<td>0.38</td>
</tr>
<tr>
<td>t1423</td>
<td>60.7</td>
<td>5.75%</td>
<td>10.9</td>
<td>3.59%</td>
<td>3.17</td>
<td>0.62</td>
<td>0.29</td>
</tr>
<tr>
<td>t9234</td>
<td>25.5</td>
<td>16.8%</td>
<td>161.1</td>
<td>11.8%</td>
<td>179.5</td>
<td>0.70</td>
<td>1.11</td>
</tr>
<tr>
<td>t13207</td>
<td>37.9</td>
<td>10.2%</td>
<td>252.2</td>
<td>2.01%</td>
<td>317.0</td>
<td>0.20</td>
<td>1.26</td>
</tr>
<tr>
<td>t382</td>
<td>8.78</td>
<td>23.6%</td>
<td>2.24</td>
<td>8.63%</td>
<td>2.71</td>
<td>0.37</td>
<td>1.21</td>
</tr>
<tr>
<td>t444</td>
<td>11.5</td>
<td>17.9%</td>
<td>2.44</td>
<td>6.27%</td>
<td>2.75</td>
<td>0.35</td>
<td>1.13</td>
</tr>
<tr>
<td>t526</td>
<td>10.6</td>
<td>11.5%</td>
<td>2.09</td>
<td>2.57%</td>
<td>1.46</td>
<td>0.22</td>
<td>0.70</td>
</tr>
<tr>
<td>t953</td>
<td>15.2</td>
<td>30.4%</td>
<td>11.5</td>
<td>3.57%</td>
<td>5.50</td>
<td>0.12</td>
<td>0.48</td>
</tr>
<tr>
<td>t1423</td>
<td>50.7</td>
<td>8.5%</td>
<td>27.9</td>
<td>8.16%</td>
<td>66.3</td>
<td>0.97</td>
<td>2.38</td>
</tr>
<tr>
<td>t9234</td>
<td>45.2</td>
<td>17.6%</td>
<td>715.1</td>
<td>9.45%</td>
<td>597.5</td>
<td>0.54</td>
<td>0.84</td>
</tr>
<tr>
<td>t13207</td>
<td>36.5</td>
<td>1.42%</td>
<td>205.9</td>
<td>0.66%</td>
<td>122.4</td>
<td>0.46</td>
<td>0.59</td>
</tr>
<tr>
<td>averages</td>
<td>13.62%</td>
<td>5.20%</td>
<td>0.44</td>
<td>0.98</td>
<td>0.44</td>
<td>0.98</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: SIZE-HINSBERGER-KOLLA results at $T_L$
slacks with small positive values, which is what we are seeking in the optimization. We might consider following Algorithm SIZE-UP with a call to Algorithm SIZE-DOWN, however, our informal experiments suggest that a single call to Algorithm SIZE-DOWN produces better results.

The second observation we make is with respect to the CPU times shown in Table 5.6 and Table 5.8. For Algorithm SIZE-UP, we observe that cross-latch optimizations (ALAP) generally require less time than the corresponding inter-latch optimizations, but that the reverse is true for Algorithm SIZE-DOWN. Previously, we had observed that cross-latch optimizations were faster for Algorithms SIZE-TILOS and SIZE-HINSBERGER-KOLLA. The different behavior of SIZE-DOWN can be easily explained by considering the direction of optimization of each algorithm. Algorithm SIZE-DOWN optimizes downward from a large initial solution; the other algorithms optimize up from a small initial circuit. Since the cross-latch solution is smaller than the corresponding inter-latch solution, Algorithm SIZE-DOWN must work harder to reach the smaller

<table>
<thead>
<tr>
<th>circuit</th>
<th>$T_H$</th>
<th>$\Delta A_{\text{SIMPLE}}$</th>
<th>$CPU_{\text{SIMPLE}}$</th>
<th>$\Delta A_{\text{ALAP}}$</th>
<th>$CPU_{\text{ALAP}}$</th>
<th>$\frac{\Delta A_{\text{ALAP}}}{\Delta A_{\text{SIMPLE}}}$</th>
<th>$CPU_{\text{ALAP}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>6.50</td>
<td>22.18%</td>
<td>0.164</td>
<td>16.07%</td>
<td>0.156</td>
<td>0.72</td>
<td>0.95</td>
</tr>
<tr>
<td>s444</td>
<td>6.88</td>
<td>29.75%</td>
<td>0.191</td>
<td>20.58%</td>
<td>0.180</td>
<td>0.69</td>
<td>0.94</td>
</tr>
<tr>
<td>s526</td>
<td>5.68</td>
<td>23.95%</td>
<td>0.285</td>
<td>11.98%</td>
<td>0.211</td>
<td>0.50</td>
<td>0.74</td>
</tr>
<tr>
<td>s953</td>
<td>7.46</td>
<td>56.66%</td>
<td>0.644</td>
<td>39.82%</td>
<td>0.496</td>
<td>0.70</td>
<td>0.77</td>
</tr>
<tr>
<td>s1423</td>
<td>35.3</td>
<td>24.69%</td>
<td>1.25</td>
<td>20.59%</td>
<td>1.09</td>
<td>0.83</td>
<td>0.87</td>
</tr>
<tr>
<td>s9234</td>
<td>26.0</td>
<td>20.08%</td>
<td>7.78</td>
<td>16.11%</td>
<td>7.72</td>
<td>0.80</td>
<td>0.99</td>
</tr>
<tr>
<td>s13207</td>
<td>26.0</td>
<td>15.52%</td>
<td>11.1</td>
<td>14.17%</td>
<td>11.26</td>
<td>0.91</td>
<td>1.01</td>
</tr>
<tr>
<td>t382</td>
<td>9.50</td>
<td>6.52%</td>
<td>0.234</td>
<td>3.38%</td>
<td>0.207</td>
<td>0.52</td>
<td>0.88</td>
</tr>
<tr>
<td>t444</td>
<td>8.93</td>
<td>8.98%</td>
<td>0.344</td>
<td>5.73%</td>
<td>0.300</td>
<td>0.64</td>
<td>0.87</td>
</tr>
<tr>
<td>t526</td>
<td>8.33</td>
<td>9.44%</td>
<td>0.406</td>
<td>2.12%</td>
<td>0.262</td>
<td>0.22</td>
<td>0.65</td>
</tr>
<tr>
<td>t953</td>
<td>10.8</td>
<td>23.42%</td>
<td>0.719</td>
<td>13.23%</td>
<td>0.543</td>
<td>0.56</td>
<td>0.76</td>
</tr>
<tr>
<td>t1423</td>
<td>63.0</td>
<td>17.18%</td>
<td>1.19</td>
<td>14.63%</td>
<td>1.08</td>
<td>0.85</td>
<td>0.91</td>
</tr>
<tr>
<td>t9234</td>
<td>34.2</td>
<td>19.14%</td>
<td>7.08</td>
<td>15.82%</td>
<td>7.06</td>
<td>0.83</td>
<td>1.00</td>
</tr>
<tr>
<td>t13207</td>
<td>47.1</td>
<td>11.74%</td>
<td>11.0</td>
<td>9.88%</td>
<td>12.47</td>
<td>0.84</td>
<td>1.14</td>
</tr>
<tr>
<td>d382</td>
<td>13.0</td>
<td>19.50%</td>
<td>0.472</td>
<td>15.34%</td>
<td>0.374</td>
<td>0.79</td>
<td>0.79</td>
</tr>
<tr>
<td>d444</td>
<td>13.8</td>
<td>26.41%</td>
<td>0.543</td>
<td>19.89%</td>
<td>0.410</td>
<td>0.75</td>
<td>0.76</td>
</tr>
<tr>
<td>d526</td>
<td>11.4</td>
<td>21.60%</td>
<td>0.641</td>
<td>12.00%</td>
<td>0.492</td>
<td>0.56</td>
<td>0.77</td>
</tr>
<tr>
<td>d953</td>
<td>14.9</td>
<td>43.77%</td>
<td>1.414</td>
<td>28.73%</td>
<td>1.11</td>
<td>0.66</td>
<td>0.79</td>
</tr>
<tr>
<td>d1423</td>
<td>70.7</td>
<td>22.66%</td>
<td>2.68</td>
<td>17.72%</td>
<td>2.26</td>
<td>0.78</td>
<td>0.84</td>
</tr>
<tr>
<td>d9234</td>
<td>52.0</td>
<td>19.62%</td>
<td>16.6</td>
<td>15.68%</td>
<td>16.0</td>
<td>0.80</td>
<td>0.96</td>
</tr>
<tr>
<td>d13207</td>
<td>51.9</td>
<td>14.69%</td>
<td>24.8</td>
<td>9.28%</td>
<td>26.6</td>
<td>0.63</td>
<td>1.07</td>
</tr>
<tr>
<td>averages</td>
<td>21.20%</td>
<td>15.23%</td>
<td></td>
<td></td>
<td>0.70</td>
<td>0.88</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: SIZE-UP results at $T_H$
cross-latch solution. Similarly, the other algorithms require more time to reach the larger inter-
latch optimized solution.

Although they use a simplified delay model, Algorithms SIZE-UP and SIZE-DOWN are
still of interest because of their linear running times, a significant advantage for large circuits.
Also, as the influence of wiring parasitics grows, the simplifying assumption they use becomes
more realistic. Both algorithms assume that variations in input capacitance are negligible com-
pared to interconnect capacitances. Thus we expect that as wiring capacitance increases, the mini-
mum cycle time attainable using Algorithms SIZE-UP and SIZE-DOWN will approach those
attainable using SIZE-TILOS and SIZE-HINSBERGER-KOLLA. Table 5.10 shows the minimum
cycle times for the original (edge-triggered) benchmark circuits s526 and s13207 attainable using
each of these algorithms where the wiring capacitance is multiplied by the scaling factor $N_{wire}$.

<table>
<thead>
<tr>
<th>circuit</th>
<th>$A_{min}$</th>
<th>$T_L$</th>
<th>$\Delta A_{SIMPLE}$</th>
<th>$\Delta A_{ACT2}$</th>
<th>$CPU_{ACT2}$</th>
<th>$\frac{\Delta A_{ACT2}}{\Delta A_{SIMPLE}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>245.75</td>
<td>4.37</td>
<td>125.23%</td>
<td>69.48%</td>
<td>0.238</td>
<td>0.55</td>
</tr>
<tr>
<td>s444</td>
<td>267.25</td>
<td>4.58</td>
<td>135.17%</td>
<td>81.20%</td>
<td>0.289</td>
<td>0.60</td>
</tr>
<tr>
<td>s526</td>
<td>321.5</td>
<td>4.29</td>
<td>112.75%</td>
<td>53.11%</td>
<td>0.351</td>
<td>0.47</td>
</tr>
<tr>
<td>s953</td>
<td>525.5</td>
<td>5.88</td>
<td>144.39%</td>
<td>82.16%</td>
<td>0.789</td>
<td>0.57</td>
</tr>
<tr>
<td>s1423</td>
<td>1029.75</td>
<td>34.4</td>
<td>119.16%</td>
<td>45.62%</td>
<td>1.73</td>
<td>0.38</td>
</tr>
<tr>
<td>s9234</td>
<td>623.5</td>
<td>17.3</td>
<td>165.12%</td>
<td>44.48%</td>
<td>11.5</td>
<td>0.27</td>
</tr>
<tr>
<td>s13207</td>
<td>9960.5</td>
<td>18.2</td>
<td>148.96%</td>
<td>18.76%</td>
<td>15.5</td>
<td>0.13</td>
</tr>
<tr>
<td>t382</td>
<td>620.75</td>
<td>4.87</td>
<td>49.58%</td>
<td>31.45%</td>
<td>0.449</td>
<td>0.63</td>
</tr>
<tr>
<td>t444</td>
<td>784.75</td>
<td>4.72</td>
<td>46.03%</td>
<td>37.46%</td>
<td>0.633</td>
<td>0.81</td>
</tr>
<tr>
<td>t526</td>
<td>767.75</td>
<td>4.72</td>
<td>47.22%</td>
<td>19.05%</td>
<td>0.562</td>
<td>0.40</td>
</tr>
<tr>
<td>t953</td>
<td>1020.5</td>
<td>6.96</td>
<td>74.35%</td>
<td>48.85%</td>
<td>1.09</td>
<td>0.66</td>
</tr>
<tr>
<td>t1423</td>
<td>1408.5</td>
<td>6.11</td>
<td>87.11%</td>
<td>51.54%</td>
<td>1.74</td>
<td>0.59</td>
</tr>
<tr>
<td>t9234</td>
<td>7585.25</td>
<td>17.2</td>
<td>135.65%</td>
<td>56.90%</td>
<td>11.29</td>
<td>0.42</td>
</tr>
<tr>
<td>t13207</td>
<td>12476.8</td>
<td>31.2</td>
<td>118.92%</td>
<td>10.63%</td>
<td>16.7</td>
<td>0.09</td>
</tr>
<tr>
<td>d382</td>
<td>559.7</td>
<td>11.5</td>
<td>110.11%</td>
<td>65.79%</td>
<td>0.59</td>
<td>0.60</td>
</tr>
<tr>
<td>d444</td>
<td>602.75</td>
<td>12.02%</td>
<td>77.49%</td>
<td>0.727</td>
<td>0.65</td>
<td></td>
</tr>
<tr>
<td>d526</td>
<td>710.5</td>
<td>8.58</td>
<td>102.04%</td>
<td>41.03%</td>
<td>0.781</td>
<td>0.40</td>
</tr>
<tr>
<td>d953</td>
<td>1343.5</td>
<td>11.8</td>
<td>112.95%</td>
<td>39.90%</td>
<td>1.74</td>
<td>0.35</td>
</tr>
<tr>
<td>d1423</td>
<td>2224.5</td>
<td>49.1</td>
<td>110.32%</td>
<td>29.10%</td>
<td>3.74</td>
<td>0.26</td>
</tr>
<tr>
<td>d9234</td>
<td>12770.5</td>
<td>28.6</td>
<td>161.14%</td>
<td>43.94%</td>
<td>24.89</td>
<td>0.27</td>
</tr>
<tr>
<td>d13207</td>
<td>21061.36</td>
<td>36.5</td>
<td>169.39%</td>
<td>19.37%</td>
<td>28.3</td>
<td>0.11</td>
</tr>
<tr>
<td>averages</td>
<td>112.36%</td>
<td>42.42%</td>
<td>0.44</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: SIZE-UP results at $T_L$
implies that the original TI delay estimates are used. The minimum cycle time for SIZE-TILOS was obtained by iteratively resizing parts until no more parts on a critical path could be resized. The minimum cycle time for SIZE-HINSBERGER-KOLLA was obtained by repeatedly minimizing the delay of the most critical path in the circuit until the critical path delay could no longer be reduced. Minimum cycle times for SIZE-UP and SIZE-DOWN were obtained by using the largest variants of each part. Columns labelled SIZE-TILOS, and SIZE-HK, and SIZE-UP/DOWN contain these cycle times, respectively. The columns labelled TILOS% and HK% contain the percentage by which the minimum cycle time for SIZE-UP/DOWN exceeds those of SIZE-TILOS and SIZE-HINSBERGER-KOLLA, respectively. We observe that increasing \( N_{\text{wire}} \) causes these percentages to decrease and that reducing \( N_{\text{wire}} \) causes the differences to increase. 

<table>
<thead>
<tr>
<th>circuit</th>
<th>( A_{\text{min}} )</th>
<th>( T_H )</th>
<th>( \Delta A_{\text{SIMPLE}} )</th>
<th>( CPU_{\text{SIMPLE}} )</th>
<th>( \Delta A_{\text{ALAP}} )</th>
<th>( CPU_{\text{ALAP}} )</th>
<th>( \Delta A_{\text{ALAP}} )</th>
<th>( CPU_{\text{ALAP}} )</th>
<th>( CPU_{\text{ALAP}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>245.75</td>
<td>6.50</td>
<td>15.97%</td>
<td>0.179</td>
<td>10.68%</td>
<td>0.175</td>
<td>0.67</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>s444</td>
<td>267.25</td>
<td>6.88</td>
<td>27.50%</td>
<td>0.199</td>
<td>18.33%</td>
<td>0.219</td>
<td>0.67</td>
<td>1.10</td>
<td></td>
</tr>
<tr>
<td>s526</td>
<td>321.5</td>
<td>5.68</td>
<td>17.03%</td>
<td>0.273</td>
<td>8.71%</td>
<td>0.297</td>
<td>0.51</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>s953</td>
<td>525.5</td>
<td>7.46</td>
<td>39.53%</td>
<td>0.594</td>
<td>25.10%</td>
<td>0.625</td>
<td>0.63</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>s1423</td>
<td>1029.75</td>
<td>35.3</td>
<td>11.17%</td>
<td>1.16</td>
<td>10.05%</td>
<td>1.11</td>
<td>0.90</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>s9234</td>
<td>6231.5</td>
<td>26.0</td>
<td>15.20%</td>
<td>7.95</td>
<td>15.52%</td>
<td>8.40</td>
<td>0.89</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>s13207</td>
<td>9960.5</td>
<td>26.0</td>
<td>5.40%</td>
<td>11.07</td>
<td>5.01%</td>
<td>12.0</td>
<td>0.93</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td>t382</td>
<td>620.75</td>
<td>9.50</td>
<td>4.87%</td>
<td>0.254</td>
<td>1.65%</td>
<td>0.277</td>
<td>0.34</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>t444</td>
<td>784.75</td>
<td>8.93</td>
<td>8.67%</td>
<td>0.324</td>
<td>3.85%</td>
<td>0.359</td>
<td>0.44</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>t526</td>
<td>767.75</td>
<td>8.33</td>
<td>7.13%</td>
<td>0.328</td>
<td>1.69%</td>
<td>0.363</td>
<td>0.24</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>t953</td>
<td>1020.5</td>
<td>10.8</td>
<td>16.07%</td>
<td>0.594</td>
<td>6.32%</td>
<td>0.652</td>
<td>0.39</td>
<td>1.10</td>
<td></td>
</tr>
<tr>
<td>t1423</td>
<td>1408.5</td>
<td>63.0</td>
<td>7.69%</td>
<td>1.08</td>
<td>7.33%</td>
<td>1.14</td>
<td>0.95</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>t9234</td>
<td>7585.25</td>
<td>34.2</td>
<td>13.05%</td>
<td>6.71</td>
<td>10.04%</td>
<td>7.31</td>
<td>0.77</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>t13207</td>
<td>12476.8</td>
<td>47.1</td>
<td>4.51%</td>
<td>10.64</td>
<td>3.96%</td>
<td>13.3</td>
<td>0.88</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td>d382</td>
<td>559.13</td>
<td>13.0</td>
<td>13.95%</td>
<td>0.453</td>
<td>9.08%</td>
<td>0.461</td>
<td>0.65</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>d444</td>
<td>602</td>
<td>13.8</td>
<td>24.34%</td>
<td>0.496</td>
<td>15.82%</td>
<td>0.543</td>
<td>0.65</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>d526</td>
<td>710.5</td>
<td>11.4</td>
<td>15.27%</td>
<td>0.602</td>
<td>6.76%</td>
<td>0.656</td>
<td>0.44</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>d953</td>
<td>1343.5</td>
<td>14.9</td>
<td>30.00%</td>
<td>1.18</td>
<td>16.11%</td>
<td>1.26</td>
<td>0.54</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td>d1423</td>
<td>2224.5</td>
<td>70.7</td>
<td>10.34%</td>
<td>2.47</td>
<td>8.88%</td>
<td>2.54</td>
<td>0.86</td>
<td>1.03</td>
<td></td>
</tr>
<tr>
<td>d9234</td>
<td>12770.5</td>
<td>52.0</td>
<td>14.83%</td>
<td>17.26</td>
<td>12.75%</td>
<td>18.58</td>
<td>0.86</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td>d13207</td>
<td>21061.5</td>
<td>51.9</td>
<td>5.13%</td>
<td>26.2</td>
<td>4.01%</td>
<td>30.5</td>
<td>0.78</td>
<td>1.16</td>
<td></td>
</tr>
<tr>
<td>averages</td>
<td>13.73%</td>
<td>9.00%</td>
<td></td>
<td>0.67</td>
<td></td>
<td>1.08</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: SIZE-DOWN results at \( T_H \)

\( N_{\text{wire}} = 1 \) implies that the original TI delay estimates are used. The minimum cycle time for SIZE-TILOS was obtained by iteratively resizing parts until no more parts on a critical path could be resized. The minimum cycle time for SIZE-HINSBERGER-KOLLA was obtained by repeatedly minimizing the delay of the most critical path in the circuit until the critical path delay could no longer be reduced. Minimum cycle times for SIZE-UP and SIZE-DOWN were obtained by using the largest variants of each part. Columns labelled SIZE-TILOS, and SIZE-HK, and SIZE-UP/DOWN contain these cycle times, respectively. The columns labelled TILOS% and HK% contain the percentage by which the minimum cycle time for SIZE-UP/DOWN exceeds those of SIZE-TILOS and SIZE-HINSBERGER-KOLLA, respectively. We observe that increasing \( N_{\text{wire}} \) causes these percentages to decrease and that reducing \( N_{\text{wire}} \) causes the differences to increase.
We also note that the differences are smaller for the larger circuit s13207, for which the estimated interconnect capacitance is larger.

### Table 5.9: SIZE-DOWN results at $T_L$

<table>
<thead>
<tr>
<th>circuit</th>
<th>$A_{\min}$</th>
<th>$T_L$</th>
<th>$\Delta A_{\text{SIMPLE}}$</th>
<th>$\Delta A_{\text{ACT2}}$</th>
<th>$CPU_{\text{ACT2}}$</th>
<th>$\Delta A_{\text{ACT2}}/\Delta A_{\text{SIMPLE}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>245.75</td>
<td>4.37</td>
<td>125.23%</td>
<td>59.00%</td>
<td>0.215</td>
<td>0.47</td>
</tr>
<tr>
<td>s444</td>
<td>267.25</td>
<td>4.58</td>
<td>135.17%</td>
<td>75.40%</td>
<td>0.246</td>
<td>0.56</td>
</tr>
<tr>
<td>s526</td>
<td>321.54</td>
<td>4.29</td>
<td>112.75%</td>
<td>43.31%</td>
<td>0.320</td>
<td>0.38</td>
</tr>
<tr>
<td>s953</td>
<td>525.56</td>
<td>5.88</td>
<td>144.39%</td>
<td>62.61%</td>
<td>0.668</td>
<td>0.43</td>
</tr>
<tr>
<td>s1423</td>
<td>1029.75</td>
<td>34.4</td>
<td>119.16%</td>
<td>33.38%</td>
<td>1.67</td>
<td>0.28</td>
</tr>
<tr>
<td>s9234</td>
<td>6231.51</td>
<td>17.3</td>
<td>165.12%</td>
<td>32.34%</td>
<td>10.6</td>
<td>0.20</td>
</tr>
<tr>
<td>s13207</td>
<td>9960.51</td>
<td>31.2</td>
<td>118.92%</td>
<td>8.71%</td>
<td>16.6</td>
<td>0.07</td>
</tr>
<tr>
<td>t382</td>
<td>620.75</td>
<td>4.87</td>
<td>49.58%</td>
<td>25.01%</td>
<td>0.355</td>
<td>0.50</td>
</tr>
<tr>
<td>t444</td>
<td>784.75</td>
<td>5.72</td>
<td>47.22%</td>
<td>14.10%</td>
<td>0.461</td>
<td>0.30</td>
</tr>
<tr>
<td>t526</td>
<td>767.75</td>
<td>6.96</td>
<td>74.35%</td>
<td>36.75%</td>
<td>0.891</td>
<td>0.49</td>
</tr>
<tr>
<td>t953</td>
<td>1020.56</td>
<td>7.48</td>
<td>112.75%</td>
<td>36.75%</td>
<td>0.891</td>
<td>0.49</td>
</tr>
<tr>
<td>t1423</td>
<td>1408.51</td>
<td>61.1</td>
<td>87.11%</td>
<td>42.01%</td>
<td>1.54</td>
<td>0.48</td>
</tr>
<tr>
<td>t9234</td>
<td>7585.25</td>
<td>17.2</td>
<td>135.65%</td>
<td>46.69%</td>
<td>10.37</td>
<td>0.34</td>
</tr>
<tr>
<td>t13207</td>
<td>12476.83</td>
<td>31.2</td>
<td>118.92%</td>
<td>8.71%</td>
<td>16.6</td>
<td>0.07</td>
</tr>
<tr>
<td>d382</td>
<td>559.71</td>
<td>7.15</td>
<td>110.11%</td>
<td>54.92%</td>
<td>0.560</td>
<td>0.50</td>
</tr>
<tr>
<td>d444</td>
<td>602.75</td>
<td>7.59</td>
<td>120.02%</td>
<td>71.47%</td>
<td>0.586</td>
<td>0.60</td>
</tr>
<tr>
<td>d526</td>
<td>710.56</td>
<td>8.58</td>
<td>102.04%</td>
<td>30.26%</td>
<td>0.676</td>
<td>0.30</td>
</tr>
<tr>
<td>d953</td>
<td>1343.51</td>
<td>11.8</td>
<td>112.95%</td>
<td>24.40%</td>
<td>1.52</td>
<td>0.22</td>
</tr>
<tr>
<td>d1423</td>
<td>2224.54</td>
<td>49.1</td>
<td>110.32%</td>
<td>20.36%</td>
<td>3.34</td>
<td>0.18</td>
</tr>
<tr>
<td>d9234</td>
<td>12770.51</td>
<td>28.6</td>
<td>161.14%</td>
<td>29.66%</td>
<td>24.0</td>
<td>0.18</td>
</tr>
<tr>
<td>d13207</td>
<td>21061.35</td>
<td>36.5</td>
<td>169.39%</td>
<td>9.62%</td>
<td>26.6</td>
<td>0.06</td>
</tr>
<tr>
<td>averages</td>
<td></td>
<td></td>
<td>112.36%</td>
<td>32.43%</td>
<td></td>
<td>0.35</td>
</tr>
</tbody>
</table>

### Table 5.10: Attainable Minimum Cycle Times for Various Net Loading Factors

<table>
<thead>
<tr>
<th>circuit</th>
<th>$N_{wire}$</th>
<th>SIZE-TILOS</th>
<th>SIZE-HK</th>
<th>SIZE-UP/DOWN</th>
<th>TILOS%</th>
<th>HK%</th>
</tr>
</thead>
<tbody>
<tr>
<td>s526</td>
<td>0.01</td>
<td>5.36</td>
<td>5.39</td>
<td>7.21</td>
<td>34.5%</td>
<td>33.8%</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>5.50</td>
<td>5.60</td>
<td>7.32</td>
<td>33.2%</td>
<td>30.9%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>6.84</td>
<td>6.73</td>
<td>8.47</td>
<td>23.9%</td>
<td>25.8%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18.75</td>
<td>19.21</td>
<td>20.16</td>
<td>7.53%</td>
<td>4.94%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>142.31</td>
<td>142.72</td>
<td>143.73</td>
<td>0.995%</td>
<td>0.705%</td>
</tr>
<tr>
<td>s13207</td>
<td>0.01</td>
<td>24.17</td>
<td>24.03</td>
<td>26.08</td>
<td>7.89%</td>
<td>8.54%</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>24.60</td>
<td>24.58</td>
<td>26.33</td>
<td>7.01%</td>
<td>7.11%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>28.51</td>
<td>28.07</td>
<td>28.82</td>
<td>1.07%</td>
<td>2.68%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>56.63</td>
<td>56.93</td>
<td>57.51</td>
<td>1.55%</td>
<td>1.02%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>348.80</td>
<td>349.21</td>
<td>349.69</td>
<td>0.26%</td>
<td>0.14%</td>
</tr>
</tbody>
</table>
For large circuits such as s13207, there is a significant speed advantage to using the linear-time algorithms SIZE-UP and SIZE-DOWN. Table 5.11 compares the optimization times required by Algorithms SIZE-DOWN, SIZE-TILOS, and SIZE-HINSBERGER-KOLLA. For small values of $N_{\text{wire}}$, the minimum cycle time attainable by Algorithm SIZE-DOWN is significantly larger than those of SIZE-TILOS and SIZE-HINSBERGER-KOLLA; as a result, the latter algorithms find smaller solutions with less effort. However, as $N_{\text{wire}}$ increases, Algorithms SIZE-TILOS and SIZE-HINSBERGER-KOLLA have to work increasingly hard to reach the same cycle times that SIZE-DOWN begins with, and sometimes these algorithms fail to achieve similarly small circuits.

<table>
<thead>
<tr>
<th>$N_{\text{wire}}$</th>
<th>$T_c$</th>
<th>$A_{\text{DOWN}}$</th>
<th>$CPU_{\text{DOWN}}$</th>
<th>$A_{\text{TILOS}}$</th>
<th>$CPU_{\text{TILOS}}$</th>
<th>$A_{\text{HK}}$</th>
<th>$CPU_{\text{HK}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>26.1</td>
<td>9961</td>
<td>15.4</td>
<td>9961</td>
<td>7.70</td>
<td>9961</td>
<td>7.04</td>
</tr>
<tr>
<td>0.1</td>
<td>26.3</td>
<td>9968.5</td>
<td>16.0</td>
<td>9970.5</td>
<td>23.4</td>
<td>9965.25</td>
<td>7.24</td>
</tr>
<tr>
<td>1</td>
<td>28.8</td>
<td>10243</td>
<td>14.2</td>
<td>10158.5</td>
<td>314.3</td>
<td>10284.2</td>
<td>209.6</td>
</tr>
<tr>
<td>10</td>
<td>57.5</td>
<td>10686.5</td>
<td>15.4</td>
<td>10557.7</td>
<td>1115.6</td>
<td>10809</td>
<td>351.2</td>
</tr>
<tr>
<td>100</td>
<td>349.7</td>
<td>11156.5</td>
<td>13.8</td>
<td>11246.5</td>
<td>&gt;2100</td>
<td>11246.5</td>
<td>484.8</td>
</tr>
</tbody>
</table>

Table 5.11: Sizing Results for s13207 with Various Net Loading Factors

For most of the circuits and algorithms we have considered, the area reductions we have seen represent a small improvement in the total circuit area, and are computed without consideration of additional area due to placement irregularities and interconnect wiring. However, we believe that the differences we have seen are significant for the following reasons: First, in every case studied, cross-latch optimization produced better areas than inter-latch optimization. Next, since the amount of area added by inter-latch optimization was small, the possible area improvements due to cross-latch optimization were also small. The appropriate measure to compare is the relative additional area required by the alternate approaches, and in this respect, cross-latch optimization reduced the amount of additional area by approximately half. Finally, timing and area constraints in modern VLSI systems are often quite tight, making even small differences in area and cycle time significant.
5.4 CMOS Input Selection

This section considers a second problem typically solved using CPM techniques: the problem of ordering gate inputs to optimize circuit timing. A number of researchers have observed that CMOS gates typically have different delays for each part input (See Section 1.2.3.5); we seek to account for this by allocating the fastest gate inputs to the most critical paths in the circuit being designed. In the following presentation we consider a simplified version of the input selection problem, chosen to again demonstrate the differences between cross-latch optimizations and traditional inter-latch CPM algorithms.

Here the problem objective we are considering is somewhat different. Instead of optimizing area for a fixed the clock schedule, we are attempting to reorder inputs with the hope of reducing the cycle time and changing the clock schedule. Since our constraint graphs require us to specify a clock schedule during the optimization, the input ordering problem is slightly harder to extend to include cross-latch optimization. However, in Section 5.4.2 we present an algorithm which can be extended and in Section 5.4.3 we consider the results of extending the algorithm to perform cross-latch optimization.

5.4.1 Background

We can illustrate the differences between inter-latch and cross-latch orderings using the circuit section in Figure 5.12-a. The circuit shown is controlled by a two-phase clock and is synchronized with level-sensitive latches. We assume that the inputs of latches $l_1$ and $l_2$ arrive sufficiently early to allow valid output signals to begin propagating from the latches on the rising edge of the clock. We also assume that the gate $G_D$ actually has two delays, $\Delta_D = 2$ and $\Delta_D' = 3$.

Our problem is to assign these delays to the inputs of $G_D$ to minimize the overall circuit cycle time. The intuitive (and traditional) approach is to minimize the maximum delay from a $\Phi_1$ to a $\Phi_2$ latch and the delay from a $\Phi_2$ to a $\Phi_1$ latch. Using such an approach, we assign $\Delta_D' = 3$ to the path $l_3 \rightarrow l_5$ and $\Delta_D = 2$ to the path $l_4 \rightarrow l_5$. The timing for this assignment is shown in Figure 5.12-b. The minimum cycle time for the circuit (assuming a symmetric clock schedule) is $4\frac{2}{3}$. 
However, a shorter cycle time is possible using the opposite assignment, shown in Figure 5.12-c. There are two paths to latch $l_5$, $l_1 \rightarrow l_3 \rightarrow l_5$ and $l_2 \rightarrow l_4 \rightarrow l_5$, and the timings of both paths are shown in Figure 5.12. If we instead assign the larger delay to the path $l_4 \rightarrow l_5$ then the delays of the paths are equalized and the minimum cycle time (again for a symmetric clock schedule) is reduced to 4. The ordering of Figure 5.12-c can be obtained by sorting the delays of $G_D$ to maximize the slack in the late signal constraint graph that corresponds to the circuit of Figure 5.12 clocked at a cycle time of 4. A lower bound on the cycle time for the circuit can be obtained from a minimum cycle time analysis (Chapter IV) with all delays through $G_D$ set to the minimum possible value $\Delta_D = 2$. Generating the late signal constraint graph for this cycle time, we see that arc $G_C \rightarrow G_D$ has a float of 1 and that arc $D_3 \rightarrow G_D$ has a float of zero. Assigning the extra delay to arc $G_C \rightarrow G_D$, reduces all floats to zero but allows the cycle time of 4 to remain feasible.

![Example Circuit](image)

**Figure 5.12: Input Ordering Example**
As mentioned previously, there is a major difference between the input ordering problem and the part selection problem of Section 5.3. In the part selection problem, the clock schedule was fixed and part sizes were varied to optimize area. In the input ordering problem, we do not know the true minimum clock schedule; instead we must rely on estimates and attempt to minimize slack in constraint graphs constructed using these estimates. In Section 5.4.3 we explore the use of three different estimates of the cycle time, one which is computed assuming that all delays are at their minimum possible value, one which assumes delays are at their maximum value, and one which is the minimum cycle time attainable using inter-latch optimizations. These estimates provide one lower and two upper bounds, respectively, on the minimum cycle time attainable by sorting part delays while allowing for cross-latch effects.

Because we did not have access to a part library with differing input-to-output delays, it was necessary to estimate the effects of input ordering on part delay. A transistor-level description of a typical simple CMOS gate is shown in Figure 5.13-a. An RC model of the pull-down chain is shown in Figure 5.13-b. Each transistor is assumed to have the equivalent resistance $R_{\text{on}}$ when switched on, transistor switching is modeled with ideal switches $S_i$, and $C_p$ is the parasitic capacitance associated with a source or drain connection to an individual transistor. The gate output drives an additional load capacitance $C_L$. 

Figure 5.13: $n$-input CMOS NAND gate
For simplicity, we assume that all the transistors in the gate are of approximately the same size and have approximately equal drive strengths. With these assumptions, the worst-case propagation delay of the NAND gate in Figure 5.13 is through the pull-down network. It is estimated as the time required to discharge the output node and any undischarged nodes in the transistor chain. For the sake of worst-case analysis, we assume that at the time of analysis, all the transistors in the chain but one have switched on and that all capacitors connected to ground are fully discharged. When the remaining transistor switches, the path from the load to ground is connected and the load capacitance can discharge along with any undischarged parasitic capacitances in the chain.

The time required to discharge the capacitances associated with the switching of switch $S_i$ is estimated with the following equation:

$$
\Delta_i = nR_{on} \left( (n+1) C_p + C_L \right) + (n-1) R_{on} 2C_p + \ldots + (n-i) R_{on} 2C_p
$$

(8)

Collecting terms gives:

$$
\Delta_i = nR_{on} \left( (n+1) C_p + C_L \right) + 2R_{on} C_p \sum_{j=1}^{i} (n-j)
$$

(9)

Replacing the summation gives:

$$
\Delta_i = nR_{on} \left( (n+1) C_p + C_L \right) + R_{on} C_p \left( 2n - i - 1 \right) i
$$

(10)

and the ratio of the delays $\Delta_i/\Delta_0$ is thus:

$$
\Delta_i/\Delta_0 = 1 + \frac{2n - i - 1}{n} \frac{C_p}{(n+1) C_p + C_L}
$$

(11)

The ratio of largest to smallest delay is:

$$
\Delta_{n-1}/\Delta_0 = 1 + (n-1) \frac{C_p}{(n+1) C_p + C_L}
$$

(12)
For our experiments, we take the worst-case delays $\Delta_{n-1}$ from the TI 1-μ library [135]. To estimate the parasitic capacitance $C_p$, we use equation (5) and the characteristics of the TI library to estimate $C_p$ as

$$C_p = \frac{1}{(n+1)} \frac{\tau_i}{R_{out,i}}.$$

### 5.4.2 Input Ordering Algorithm

As mentioned previously, a lower bound on the minimum cycle time can be obtained by assuming that the delay through each part is $\Delta_0$, the delay of the fastest part input. In this section, we present an algorithm which begins with this initial estimate and re-inserts the additional delays into the network to minimize their impact on the cycle time.

The algorithm for sorting part inputs (Algorithm SORT-INPUTS) is shown in Figure 5.14. We begin by analyzing the circuit with all delays at their minimum possible values; that is, we assume that all signals propagate through the fastest input for each gate. We then compute slacks and floats in the network and work backwards, greedily allocating the inputs with largest delay to the paths with the largest (most positive) slacks. The sorting procedure makes a single pass through the constraint network, and if the number of inputs to each gate is bounded by a constant, Algorithm SORT-INPUTS runs in linear time.

```
generate constraint graph using minimum input-to-output delays
compute actual event times for all nodes in the graph
for each node V in reverse topological order
    compute required time at node V
    if V corresponds to a multi-input gate
        calculate floats on input arcs of V
        sort input arcs of V by floats
        assign largest delays to arcs with largest floats
    end if
end for
```

Figure 5.14: Algorithm SORT-INPUTS

Extending this algorithm to cross-latch optimization is straightforward. A cyclic constraint graph is first generated, then arcs are again removed to eliminate cycles. There are a few difficulties which arise, however. First, the construction of our constraint graphs requires that we specify a target cycle time for our circuit, but since we are attempting to reorder inputs to minimize the cycle
time, the actual minimum is unknown. We can place a lower bound on the cycle time, $T_{\text{min}}$, by noting that the optimized circuit will not run faster than an equivalent circuit in which the delay through each gate is set to its minimum possible value. An upper bound, $T_{\text{max}}$, can similarly be obtained by computing the minimum cycle time for the circuit when all gate delays are at their maximum values. However, a better upper bound is the minimum cycle time of the circuit obtained by inter-latch optimization, $T_{\text{inter}}$. Since the solution space defined by inter-latch optimization is a subset of the solutions allowed in the cross-latch optimization approach, we know that the cross-latch solution should always be at least as good as the inter-latch solution. Recall that whenever the time shifts in the circuit are all equal, the minimum cycle time attainable using inter-latch optimization can be obtained independently of the cycle time value used to construct the constraint graph. Experiments using each of these times as the initial cycle time are presented in the following section.

The second problem is that at the target cycle times we specify, we cannot guarantee that breaking loops using the ALAP or ASAP methods will produce timing-feasible constraint graphs. As a result, we use the ACTUAL arc-breaking method to remove cycles in the constraint graphs.

5.4.3 Experimental Results

Table 5.12 lists the results of experiments using Algorithm SORT-INPUTS on the benchmark circuits of Table 5.1. The table shows $T_{\text{best}}$ and $T_{\text{worst}}$; these are the cycle times of each circuit assuming each gate delay is equal to its minimum and maximum input-to-output delay, respectively. $T_{\text{cross(best)}}$ and $T_{\text{cross(worst)}}$ are the minimum cycle times after running Algorithm SORT-INPUTS on a late-signal constraint graph, where loops are broken using the ACTUAL criterion based on timing analyses performed at $T_{\text{best}}$ and $T_{\text{worst}}$, respectively. In both cases, we do not know the true minimum attainable cycle time using cross-latch optimization; $T_{\text{best}}$ and $T_{\text{worst}}$ are simply estimates that are used to construct the constraint graph. The next columns show $T_{\text{inter}}$ and $T_{\text{cross(inter)}}$; $T_{\text{inter}}$ is the minimum cycle time using inter-latch optimization, and $T_{\text{cross(inter)}}$ is the minimum cycle time obtained using $T_{\text{inter}}$ as the initial target cycle time. The remaining col-
columns show the ratios of each inter-latch cycle time with the minimum cycle time using cross-latch optimization.

In the initial set of circuits we examined, the cross-latch optimizations generally produced either a moderate or no cycle time reduction over the simpler inter-latch optimization. Because of the linear time complexity of Algorithm SORT-INPUTS, run times for all sorts were very small, under 15 CPU seconds for even the largest examples. Because they considered more complex constraint graphs, the run times for cross-latch optimizations were slightly larger than those of inter-latch optimizations, but again, in all cases CPU times were less than 15 seconds. Of the three starting points, using $T_{inter}$ produced slightly better results, but as shown by the averages, the differences were not significant. However, in a few cases, the cross-latch optimization resulted in a larger cycle time than that obtained using inter-latch optimization. This was due to the differences between the initial cycle time and the true optimal cycle time attainable using input reordering. Note that when $T_{inter}$ was used as the starting point, the cross-latch optimization never resulted in a larger cycle time than $T_{inter}$, so that $\frac{T_{cross(inter)}}{T_{inter}} \leq 100\%$.

Table 5.13 lists the results of the same procedures for the benchmark circuits after using Algorithm SIZE-HINSBERGER-KOLLA to equalize path delays. For these circuits, the differences in cross-latch and inter-latch optimization are slightly larger, but probably not significantly. We again note that solutions obtained using $T_{best}$ and $T_{worst}$ as starting points occasionally produce cycle times greater than $T_{inter}$, but that again, $\frac{T_{cross(inter)}}{T_{inter}} \leq 100\%$. It is also interesting to note that for these examples, on average, using $T_{worst}$ as the starting point produced slightly lower cycle times.

Although the results seen here show a relatively small difference between inter-latch and cross-latch optimizations, when we increase the relative differences among input delays, these differences grow more pronounced. Table 5.14 shows the results of inter-latch and cross-latch optimization where the differences in gate input-to-output delays have been exaggerated. For each row in the table, adjacent path delays through a gate are assumed to differ by a factor of $(1 + DF)$, where
is called the delay factor. Cross-latch optimization was run using the same three starting points as before; the results for each starting point were identical, and are listed under the headings $T_{\text{cross(best)}}$ and $T_{\text{cross(worst)}}$. As the table shows, as the delay factor increases, the potential cycle time savings (due to $T_{\text{best}}$) also increases, and the differences between cross-latch and inter-latch optimization grow.

### 5.5 Conclusions

This chapter has presented initial applications of the extended formulation of Chapter III to a pair of CPM-based optimizations. To demonstrate its effectiveness, we applied the extended model to two problems, the part selection problem and the CMOS input ordering problem. For
both problems, we observed modest but distinct improvements in the results obtained when the extended formulation for cross-latch optimization was used.

### Table 5.13: Input Sorting Results (after sizing to equalize delays)

<table>
<thead>
<tr>
<th>circuit</th>
<th>$T_{\text{best}}$</th>
<th>$T_{\text{cross(best)}}$</th>
<th>$T_{\text{worst}}$</th>
<th>$T_{\text{cross(worst)}}$</th>
<th>$T_{\text{inter}}$</th>
<th>$T_{\text{cross(best)}}$</th>
<th>$T_{\text{cross(worst)}}$</th>
<th>$T_{\text{inter}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>5.51</td>
<td>5.69</td>
<td>6.30</td>
<td>5.69</td>
<td>5.82</td>
<td>5.69</td>
<td>97.77%</td>
<td>97.77%</td>
</tr>
<tr>
<td>s444</td>
<td>5.94</td>
<td>5.99</td>
<td>6.51</td>
<td>5.99</td>
<td>6.12</td>
<td>5.99</td>
<td>97.88%</td>
<td>97.88%</td>
</tr>
<tr>
<td>s526</td>
<td>5.92</td>
<td>5.96</td>
<td>6.46</td>
<td>5.96</td>
<td>5.96</td>
<td>5.96</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>s953</td>
<td>6.91</td>
<td>6.95</td>
<td>7.61</td>
<td>6.96</td>
<td>6.91</td>
<td>6.91</td>
<td>100.58%</td>
<td>100.72%</td>
</tr>
<tr>
<td>s1423</td>
<td>27.22</td>
<td>27.27</td>
<td>34.28</td>
<td>27.27</td>
<td>27.41</td>
<td>27.27</td>
<td>99.49%</td>
<td>99.49%</td>
</tr>
<tr>
<td>s9234</td>
<td>19.63</td>
<td>20.17</td>
<td>21.42</td>
<td>20.17</td>
<td>20.17</td>
<td>20.17</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>s13207</td>
<td>22.08</td>
<td>22.23</td>
<td>24.04</td>
<td>22.23</td>
<td>22.87</td>
<td>22.23</td>
<td>97.20%</td>
<td>97.20%</td>
</tr>
<tr>
<td>t382</td>
<td>9.17</td>
<td>9.17</td>
<td>9.55</td>
<td>9.17</td>
<td>9.42</td>
<td>9.17</td>
<td>97.35%</td>
<td>97.35%</td>
</tr>
<tr>
<td>t444</td>
<td>9.71</td>
<td>9.71</td>
<td>9.94</td>
<td>9.71</td>
<td>9.71</td>
<td>9.71</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>t526</td>
<td>9.27</td>
<td>9.35</td>
<td>9.93</td>
<td>9.35</td>
<td>9.41</td>
<td>9.35</td>
<td>99.36%</td>
<td>99.36%</td>
</tr>
<tr>
<td>t953</td>
<td>10.28</td>
<td>10.28</td>
<td>11.45</td>
<td>10.28</td>
<td>10.54</td>
<td>10.28</td>
<td>97.53%</td>
<td>97.53%</td>
</tr>
<tr>
<td>t1423</td>
<td>50.08</td>
<td>50.25</td>
<td>62.14</td>
<td>50.25</td>
<td>50.50</td>
<td>50.25</td>
<td>99.50%</td>
<td>99.50%</td>
</tr>
<tr>
<td>t9234</td>
<td>23.71</td>
<td>23.87</td>
<td>25.47</td>
<td>23.87</td>
<td>24.19</td>
<td>23.87</td>
<td>98.68%</td>
<td>98.68%</td>
</tr>
<tr>
<td>t13207</td>
<td>37.40</td>
<td>37.40</td>
<td>37.86</td>
<td>37.40</td>
<td>37.40</td>
<td>37.40</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>d382</td>
<td>7.84</td>
<td>7.90</td>
<td>8.70</td>
<td>7.93</td>
<td>7.99</td>
<td>7.90</td>
<td>98.87%</td>
<td>98.87%</td>
</tr>
<tr>
<td>d444</td>
<td>9.21</td>
<td>9.45</td>
<td>10.11</td>
<td>9.50</td>
<td>9.65</td>
<td>9.56</td>
<td>97.93%</td>
<td>98.45%</td>
</tr>
<tr>
<td>d526</td>
<td>9.06</td>
<td>9.21</td>
<td>10.29</td>
<td>9.16</td>
<td>9.31</td>
<td>9.16</td>
<td>98.93%</td>
<td>98.39%</td>
</tr>
<tr>
<td>d953</td>
<td>13.82</td>
<td>13.90</td>
<td>15.22</td>
<td>13.99</td>
<td>13.82</td>
<td>13.82</td>
<td>100.58%</td>
<td>101.23%</td>
</tr>
<tr>
<td>d1423</td>
<td>40.62</td>
<td>40.92</td>
<td>50.73</td>
<td>40.76</td>
<td>40.92</td>
<td>40.92</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>d9234</td>
<td>43.48</td>
<td>44.44</td>
<td>47.12</td>
<td>44.42</td>
<td>44.44</td>
<td>44.44</td>
<td>100.00%</td>
<td>99.95%</td>
</tr>
<tr>
<td>d13207</td>
<td>41.64</td>
<td>42.63</td>
<td>45.82</td>
<td>41.64</td>
<td>43.38</td>
<td>42.88</td>
<td>98.27%</td>
<td>95.99%</td>
</tr>
<tr>
<td>totals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99.04%</td>
<td>98.97%</td>
</tr>
</tbody>
</table>

### Table 5.14: Input Sorting Results for t953 (after sizing) with Various Delay Factors

<table>
<thead>
<tr>
<th>$DF$</th>
<th>$T_{\text{best}}$</th>
<th>$T_{\text{inter}}$</th>
<th>$T_{\text{cross}}$</th>
<th>$\frac{T_{\text{cross}}}{T_{\text{inter}}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>10.50</td>
<td>10.50</td>
<td>10.50</td>
<td>100.00%</td>
</tr>
<tr>
<td>0.2</td>
<td>9.75</td>
<td>9.94</td>
<td>9.75</td>
<td>98.09%</td>
</tr>
<tr>
<td>0.3</td>
<td>9.15</td>
<td>9.58</td>
<td>9.15</td>
<td>95.51%</td>
</tr>
<tr>
<td>0.4</td>
<td>8.74</td>
<td>9.53</td>
<td>8.74</td>
<td>91.71%</td>
</tr>
<tr>
<td>0.5</td>
<td>8.40</td>
<td>9.48</td>
<td>8.40</td>
<td>88.61%</td>
</tr>
<tr>
<td>0.6</td>
<td>8.10</td>
<td>9.44</td>
<td>8.34</td>
<td>88.35%</td>
</tr>
<tr>
<td>0.7</td>
<td>7.88</td>
<td>9.41</td>
<td>8.23</td>
<td>87.46%</td>
</tr>
<tr>
<td>0.8</td>
<td>7.73</td>
<td>9.38</td>
<td>8.15</td>
<td>86.89%</td>
</tr>
<tr>
<td>0.9</td>
<td>7.61</td>
<td>9.35</td>
<td>8.08</td>
<td>86.42%</td>
</tr>
<tr>
<td>1.0</td>
<td>7.49</td>
<td>9.32</td>
<td>8.01</td>
<td>85.94%</td>
</tr>
</tbody>
</table>

Table 5.14: Input Sorting Results for t953 (after sizing) with Various Delay Factors
We believe that many other CPM-based optimizations can be similarly extended to perform cross-latch optimization, and that this is an interesting area for future research. Other areas of research include an evaluation of the various techniques for loop breaking listed in Section 5.2 for general circuits and guidelines for their use. Also, since the loop breaking modifications produce fundamental limits on the solution quality, approaches which can be used on unmodified cyclic constraint graphs could provide additional improvement.

Despite the limitations imposed by loop-breaking, we conclude with three general benefits of the cross-latch optimization approaches described in this chapter:

1. Cross-latch optimization is simple to implement. Each algorithm we examined was formulated using general CPM networks. The formulation of Chapter III incorporates level-sensitive latch timing behavior into these CPM networks, and the extension only requires the additional step of breaking cycles in the constraint graph, which can be implemented using a simple depth-first traversal of the constraint network.

2. Cross-latch optimization produces better results. In virtually all cases we examined, the additional flexibility of cross-latch optimization allowed the optimization algorithms to find solutions of equal or better quality to those obtained using simpler inter-latch optimizations.

3. Cross latch optimization adds no significant computational cost. The only extra computation required is the inexpensive loop-breaking step, which can be performed in a depth-first traversal of the constraint graph. In several significant cases (Algorithms SIZE-TILOS, SIZE-HK, and SIZE-UP), we observed that cross-latch optimizations actually required less time than comparable inter-latch optimizations. This results from the fact that the running times of these algorithms depend on the difficulty of satisfying the constraints implied by the target clock schedule; adding accurate models of latch timing eases these constraints, allowing the algorithms to more quickly find a feasible solution.

To these we add the following limitation:
4. Cross-latch optimization requires a target clock schedule. Traditional critical path methods can minimize circuit cycle time by maximizing slack in a network; maximizing slack in a late-signal constraint graph simply increases the margin on the circuit setup constraints. This margin will have a varying influence on the cycle time, depending on the latencies of the associated paths. Slack maximization may be a useful heuristic for minimizing cycle time; however, it is preferable to optimize for a predetermined clock schedule. In contrast, inter-latch optimization does not necessarily require a target clock. If the total time shifts of all paths are equal, then the minimum cycle time can be obtained by maximizing slack regardless of the cycle time used to construct the constraint graph.
CHAPTER VI
CONCLUSIONS AND FUTURE DIRECTIONS

6.1 Research Contributions

This dissertation has presented a general model of the timing behavior of circuits with level-sensitive latches and has applied it to a variety of analysis and optimization problems. The model allows for exact and rapid static timing verification, the determination of the optimal clock signals to control a circuit, and a wide variety of timing-driven design optimizations. It has also proven to be a popular formulation; since this work began, a large number of other researchers have used this model as a framework for studying the timing of circuits with level-sensitive latches [18, 66, 69, 85, 89, 118, 119-123, 132-133].

The specific contributions of this dissertation follow the general pattern outlined in the introduction. In Chapter II, we presented a revised formulation which includes the ability to model clock skew, differing data-to-output and clock-to-output latch delays, and the full complement of possible positive and negative level-sensitive and edge-triggered devices. We also presented a graph representation of the model and an extended formulation which accounted for individual gate delays and which allowed for differing input-to-output delays for each gate.

Chapter III addressed the first major timing analysis problem: timing verification. After reviewing existing timing verification techniques, we presented definitions of three types of critical paths which arise in circuits with level-sensitive latches. We discussed the effects of each type of path on both circuit operation and timing analysis. Special difficulties appear when circuits contain synchronous feedback paths clocked with level-sensitive latches; we present methods for identifying and analyzing critical paths in the presence and absence of these loops.
In Chapter IV, we considered the problem of optimally clocking circuits with level-sensitive latches; that is, the determination of the clock schedule which produced the minimum cycle time of a circuit or which optimized some other linear function of the model parameters. We considered a number of simplifications to simplify the optimization problem, and described new procedures for optimizing single-phase pipelines and circuits with fixed-latency clock schedules. For the general case, we defined the min-max Linear Programming (mmLP) problem and used this formulation to develop branch-and-bound algorithms for the original unrestricted optimal clocking problem proposed by Sakallah, Mudge, and Olukotun [116].

Finally, in Chapter V, we applied the graph model of Chapter II and the critical path definitions of Chapter III to the problem of timing-driven design; specifically, we looked the timing-driven part-selection problem, which seeks to select part implementations from a library to minimize cost while satisfying circuit timing constraints, and the CMOS input ordering problem, which seeks to reorder the inputs of CMOS gates to improve timing performance. Our focus was on the Late Signal Constraint graphs; as a result, we were interested primarily in guaranteeing that all long path and loop constraints were satisfied. Slack values in the constraint graphs provided information about long paths; since no slack was initially available for the loop constraints, we used a simple procedure to remove selected arcs and adjust the remaining arc weights to cause the slack values to also reflect the loop constraints. Although removing these arcs limited the range of optimization we could perform, we observed distinct improvements when part selection algorithms were modified to perform cross-latch optimization; this was in contrast to simpler inter-latch optimization techniques which simply treat latches as edge-triggered devices. We also observed improvements when inter-latch optimization was used in the input ordering problem, however, these results were muted by the conservative assumptions we used to estimate variations in gate input-to-output delays. When these variations were exaggerated, the advantages of cross-latch optimization grew more significant. In nearly all cases observed, the additional flexibility of cross-latch optimizations produced superior results at negligible additional cost.
6.2 Ideas for Future Research

Again following the general outline of the thesis, we can suggest a number of unsolved problems and areas for additional research:

The timing models of Chapter II assume that a static data-independent analysis is to be performed. In many cases, significant improvements in accuracy can be made by considering data dependent effects in timing analysis. The simplest such case is that of incorporating models for differing rise and fall delays at the outputs of logic gates; more complex extensions include the modeling of the data-dependent properties of dynamic logic gates and the elimination of unsensitizable delays.

Another interesting modeling problem is the development of macromodels for specific synchronizing devices such as the D-type latches and flip-flops assumed throughout this thesis. In this work, we have assumed that all synchronizer properties are given: this includes setup times, hold times, minimum pulse widths, and input-to-output delays. In reality, these parameters must be determined by device or cell designers, often by circuit simulation. Simpler, higher-level methods may be possible that are based on the delays of components used to construct synchronizers. Initial work in this area suggests that model parameters may be obtainable using min-max Linear Programming; methods for developing the relevant constraints are an interesting area for future research, as is a comparison of the synchronizer macromodeling problem with the general problem of timing verification of asynchronous circuits [13, 98].

An unsolved problem of theoretical interest is that of proving that the general optimal clocking is NP-hard. Its non-convex structure suggests that no polynomial-time solution procedure exists, and it is an instance of a known NP-hard problem (min-max Linear Programming), but to our knowledge, the NP-hardness of optimal clocking remains unproven.

In Chapter V, we presented results of cross-latch optimizations on two timing-driven design problems; extensions to other problems would provide useful points for comparison. If a wider range of benchmarks were available, it would be useful to explore the various cycle-breaking approaches of Section 5.2 in more detail. Optimization techniques which leave cycles intact are also desirable; it would be interesting to compare the results of Section 5.3 with a sizing algorithm that used an optimal clocking algorithm to find minimum cycle times and one of the critical
path identification techniques of Chapter II to identify paths or parts to be resized. Leaving cycles in the constraint graph would no doubt add to the complexity and running time of the technique; but the relative performance improvements may make the extra cost worthwhile.

One final area which should be of primary interest is that of the delay testing of circuits which use level-sensitive latches. When circuits are edge-triggered, delay testing is relatively straightforward; it simply requires the ability to enter test patterns, step the machine at the target clock frequency, and then check the results. However, for level-sensitive circuits, critical paths may extend through an arbitrary number of transparent latches, requiring delay tests that cover multiple cycles of machine operation. Also, it may be very difficult to ensure that all loop constraints are satisfied; recall that in Chapter II we saw that an arbitrarily large number of iterations (or machine cycles) may be required for a violated loop constraint to cause a timing error. Until these problems are resolved, it will be necessary to rely on indirect methods of delay testing or, when direct delay tests are required, to revert to traditional models that treat latches as edge-triggered devices.
BIBLIOGRAPHY


