OPTIMIZING AVERAGE-CASE PERFORMANCE IN THE TECHNOLOGY MAPPING OF ASYNCHRONOUS CIRCUITS

by

Wei-Chun Chou

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

August 1999

Dedication

To my father, mother, and my wife Jean.

Acknowledgments

First, I would like to thank my research advisor, Professor Peter Beerel, for his inspired instruction of my research. I would also like to thank Professor Massoud Pedram, Douglas Ierardi, Sandeep Gupta, and Viktor Prasanna for being my defense and qualify committee for their time spent on a challenging review of my thesis. Finally, I would like to thank Professor James Ellison and Kai Hwang for helping me with their support.

Contents

List Of Tables

List Of Figures

Abstract

This thesis presents a technology mapper that optimizes average-case performance of two asynchronous design styles: burst-mode control circuits and one-hot domino control circuits. The mapping procedure consists of two phases: decomposition and covering. Circuits are first decomposed into unmapped NAND2-INV networks which are then covered using a given library of gates. The specification of these circuits are preprocessed using stochastic techniques or architectural simulations to determine the input patterns of interest and their relative frequencies which guide the mapper to shorten the critical paths for common input patterns. More specifically, the mapper minimizes the weighted sum of circuit performance for all input patterns, thereby optimizing the mapped circuits for average-case performance. Our experimental results demonstrate that, with manageable run-times, our mapped circuits have significant higher average-case performance and smaller area than the comparable circuits mapped using a leading conventional mapping technique which minimizes worst-case delay.

Chapter 1

Introduction

As desired computation frequencies exceed 1 GHz, asynchronous circuits are becoming an attractive alternative to synchronous circuits because they remove the high power consumption of clock distribution circuitry, avoid clock-skew problems, and circumvent minimum pulse-width limitations through clock trees. Recent emerging asynchronous design have demonstrated impressive results for digital signal processing [32, 87, 79] and microprocessors [26, 42, 4], but the lack of CAD support is still limiting their advances in some areas. This thesis introduces a CAD tool for two specific design methodologies: *burst-mode control circuits* and *one-hot domino circuits*. The former is a popular Huffman finite state machine (FSM) for asynchronous controllers [29]. The latter is domino-circuit-based combinational logic often used to convert data signals into control signals in microprocessors, such as instruction decoding [62]. This thesis, focusing on the technology mapping of the above two asynchronous design styles, proposes the first known technique in the literature to optimize *average-case performance* of asynchronous circuits.

1.1 Asynchronous circuits

Figure 1.1 illustrates the main difference between synchronous and asynchronous circuits. Whereas synchronous circuits use a global clock to latch and synchronize data, asynchronous circuits use handshaking signaling implemented with asynchronous controllers. The operation of the example asynchronous pipeline shown in the figure is as follows. Once data is available at the block input, the previous stage sends a request to the controller of the current stage to latch new data. When the current output data has been latched, the next stage sends an acknowledge back indicating that the old input data is no longer needed. Once

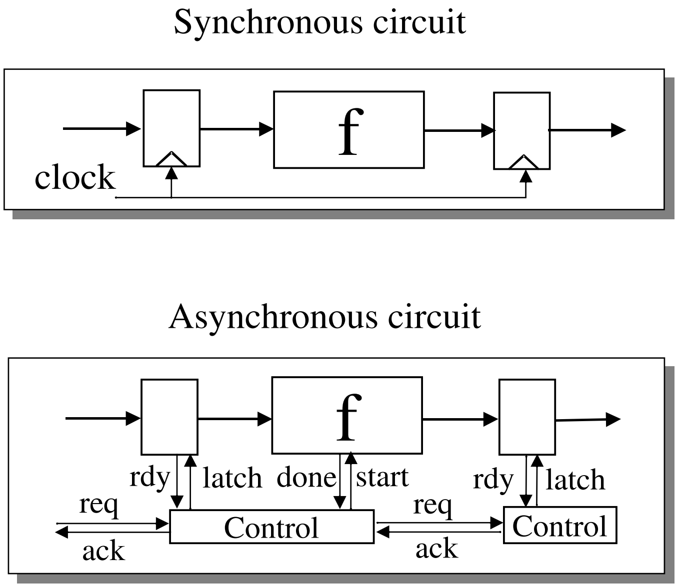

Figure 1.1: Synchronous vs. asynchronous circuits.

both of these signals arrive, the controller of the current state latches the new input data. In the most robust systems, the controller then waits for the latch to send a ready signal, *rdy*, indicating the input data has been latched. In more aggressive designs, the ready signal is not implemented and instead the latching delay is assumed to be small and fixed and is accounted for in the design of other delay lines and/or handshaking logic. Once latched, the controller simultaneously sends a start signal to the computation block and an acknowledge to the previous stage. When the computation is done, the controller sends a request to next stage. Once the next stage has been acknowledged by subsequent next stage, the next stage latches the output data and sends an acknowledge back to the current stage which can then repeat the computation cycle for next input data. Notice that in some design methodologies, e.g. *Micropipelines*, the done signal is implemented as a delayed version of the request signal that matches the longest path in the combinational logic [69]. In this way, the asynchronous circuit can use datapath blocks initially designed for use in synchronous circuits. For critical components, however, additional *completion sensing* circuitry is used to generate the done signal.

The circuit in Figure 1.1 illustrates two important features of asynchronous circuits. First, they have short and local control wires which alleviates problems arising from a long global clock wire. Second, asynchronous circuits are event-driven and consequently can respond immediately to the completion of a computation without waiting for a clock edge.

Because the computation delays are temperature, process, and data-dependent (if completion sensing circuitry is used), one advantage of asynchronous circuits is that their performance can and should be characterized by their average-case behavior, not their worst-case behavior which is the general performance metric used for synchronous circuits. Moreover, the asynchronous handshaking between components facilitates good modularity which makes large system design using asynchronous circuits relatively easy. In addition, in some asynchronous designs the handshaking removes the need for global timing assumptions which makes migrating to a new technology process significantly easier than their synchronous counterparts.

In this thesis, we stress the data-dependent feature of aggressively designed asynchronous circuits. The computation delay is data-dependent because different data propagate through different paths which cause different propagating delays in the circuit. We also notice that some input data appears commonly and some appears seldomly. In this thesis, we focus on synthesis for which our goal is to optimize the propagation paths for common input data, such that for most of the time, data propagation is fast. Or, more precisely, our goal is to optimize the *average-case* performance of the circuit. In fact, some asynchronous designs have demonstrated the potential advantages of high average performance [53, 54, 19]. The optimization of these circuits, however, is often very time-consuming and error-prone due to the lack of supporting CAD tools. As far as we know, there has been very little previous work on performance optimization. We were thus motivated to develop techniques for optimizing the average performance.

1.2 Asynchronous design methodologies

Researchers have developed many methodologies and techniques to design asynchronous circuits. Some of the techniques are limited to controller design, whereas others, are limited to data-path. A few techniques are applicable to both. The key design of asynchronous controllers is to communicate with other controllers and their associated data-paths using handshaking signals. Asynchronous data-paths, conceptually, must be designed to accept their associated controller's start signals to start computations, sense the completion of the computation, and send done signals back to their associated controllers.

In all cases, the controllers must be hazard-free which means they should not generate any runt pulses (or glitches) under any possible set of gate delays and environmental response time. Different techniques have been developed based on different models of the gate delays and the environment. A delay model that considers more possible delays with less delay assumptions will produce more general and robust asynchronous designs at the possible expense of performance. A delay model that considers fewer possible delays with more timing assumptions can often yield better circuits but must be designed more carefully in order to ensure correct operation. A similar tradeoff occurs with regards to assumptions as to how the environment interacts with the controller.

One class of asynchronous controllers are the *fundamental-mode* circuits implemented using Huffman finite state machine style consisting of combinational logic and feedback delay elements [29]. Huffman circuits are designed under *unbounded-delay model* which assumes that all gates and wires have unknown delays. It takes input data into combinational logic, generates output data, and feedbacks state variables through delay elements into combinational logic to stabilize the circuit. It has an environmental assumption that next input data cannot arrive until feedback signals have already stabilized, i.e., the circuit must obey the *fundamental-mode constraint*. Naive fundamental-mode Huffman circuits only allow single input change. More advanced fundamental-mode circuits, such as *burst-mode circuits*, allow input data arrive in any order as an *input burst*, generate an output burst, and feedback a state burst [88, 89, 18]. Nowick's UCLOCK [52] and Yun's 3D [88, 89], using hazard-free two-level minimization and state encoding algorithms, generate two different styles of burst-mode circuits.

In contrast to fundamental-mode circuits, *IO-mode* circuits, allows inputs to change at any time allowed by the specification. That is, the specification of an IO-mode circuit, graph or language based, not only dictates how the controller should work but also specifies the legal input behavior of the environment. IO-mode circuits allows for the circuit and the environment to interact with greater concurrency than typically allowed in fundamental-mode circuits.

IO-mode circuits can be further classified based on the associated gate delay model. One class of IO-mode circuits are *timed* circuits for which a lower and upper bound for each transition are given [51]. Alternatively, many controller design methodologies use the more conservative speed-independent delay model, in which gate delays are assumed unbounded but all wire delays are negligible [46]. Finally, other IO-mode circuits are designed using a similar quasi-delay-insensitive model, in which all gate and wire delays are unbounded except for specified wire forks which are assumed to be *isochronic* [41, 77], i.e., each branch of a wire fork has the same delay.

IO-mode circuits are typically specified using graph-based or language-based formalisms. For example, timed circuits are specified using *timed handshaking expansion* (HSE) which are internally translated to graph based model during the synthesis process. Speed-independent circuits are often specified using STGs and synthesized using state graphs [15, 14, 44, 2] . I-Net are sometimes used to design the delay-insensitive circuits [45]. Language-based compilation methodologies are also popular. Communicating sequential processes (CSP) and the Tangram language are two common languages used to design quasi-delay-insensitive circuits [41, 77].

For data-path design, dual-rail and bundled-data techniques are the most common alternatives. In dual-rail designs, each data bit is encoded in a two-bits, one for the true rail and one for the false rail. This allows the receiver of the signal to locally know when the data is valid. Dual-rail signaling can be used in a completely delay-insensitive datapath or within a datapath unit with timing assumptions (e.g., [90]). Alternatively, *bundle-data data-path blocks* use an additional worst-case delay line for each data-path component to implement the done signal. Hybrid approaches have also been explored. In *speculative completion sensing*, proposed by Nowick [53, 54], multiple delay lines of varying lengths that are multiplexed to improve the average-case performance of bundled data-path blocks. The control of the multiplexor is done by a separate *abort* logic. In addition, hybrid approaches that combine speculative completion sensing and traditional completion sensing have also been proposed [54].

Ivan Sutherland combines the features of quasi-delay-insensitive control circuits and the bundled-data datapath in a novel asynchronous pipeline structure, called Micropipelines [69]. He proposes a small set of control circuits that can be combined to produce most architectures of interest. The bundled-data data-path structure can have standard synchronous (hazardous) data-path blocks (e.g., adders, or multipliers) be used without fear of incorrect operation. The approach, however, has two disadvantages. First, the bundled-data constraint implies that all matched delay lines are set to the worst-case computation time. Second, building complex control structures out of the proposed small set of components yields large, slow controllers, which often are in the system critical path. Both of these disadvantages tend to imply sub-optimal performance. Consequently, in practice, more sophisticated completion-sensing-based components are used for performance critical operations and when necessary, more sophisticated controller designs are used. Nevertheless, because the naive control circuits are relatively easy to use, building complex asynchronous systems, e.g., microprocessors, is feasible and has generated promising results, as demonstrated by the AMULET group [24, 57, 26, 25].

For some applications, the boundary between data-path and control is blurred. This is true in particular for those data-path components, such as instruction decoders, whose primary responsibility is to generate control signals from data. For such components, an efficient one-hot domino design has been recently proposed to remove the completion sensing overhead from the system's critical path.

In this thesis, our focus is on high-performance burst-mode controllers and the one-hot domino circuits described above. In particular, we develop the first technology mapping techniques for these circuits which are optimized for average-case performance. We now describe these two design methodologies in more detail.

1.2.1 Burst-mode circuits

Burst-mode control circuits are typically specified in extended burst-mode (XBM) [88] and implemented using a modified Huffman architecture [52, 85, 88]. Each circuit consists of a combinational logic block with some outputs fed-back to the inputs through delay elements. The XBM specification and an implementation of SCSI-INIT-SEND controller are shown in Fig. 1.2 as an example.

In each state, the circuit waits for a set of specified input transitions, referred to as an *input burst*; upon detecting specified input transitions, it toggles a set of output signals, referred to as an *output burst*. In addition, depending on the implementation style, the circuit may internally toggle a number of state signals either concurrently with output signals or in a separate *state burst* before or after the output burst [85]. For the machine to operate properly, the environment must obey the generalized *fundamental-mode constraint* which essentially states that the next input burst must arrive *after* the fed-back signals have propagated deeply enough into the combinational logic block, in order to ensure that there are no unexpected hazards [10, 85].

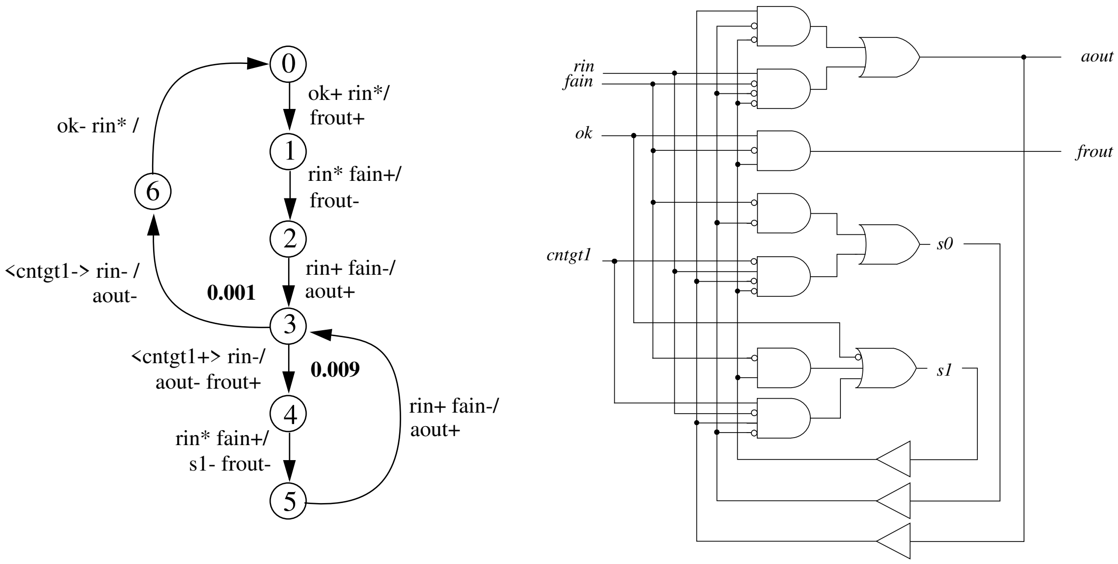

Figure 1.2: Scsi-init-send example.

1.2.2 One-hot domino circuits

As already mentioned, one-hot domino circuits can be used to implement combinational circuits that convert data signals into control signals. These circuits typically perform instruction decoding of some type and, due to their complexity, are often a bottleneck in both synchronous and asynchronous microprocessors.

Traditionally, combinational circuits that convert data into control signals are implemented using single-rail bundled-data techniques. This method unfortunately implies that the delay of the circuit is determined by the most complex data needed to be decoded (rather than the most common data). Dual-rail techniques, in which each signal is encoded with two bits, can also be used to design these circuits and facilitate the optimization for average-case performance. Traditional dual-rail designs, however, are typically larger, consume more power, and are slower (due to the complex completion sensing structures required) than single-rail designs.

We consider a different design style for these decoders which applies a combination of domino logic, dual-rail signaling, and one-hot encoded outputs. Chris Myers initially conceived of this design [49] and Benes et al. independently developed a similar technique that they used in a decompression circuit for embedded processors [4]. Domino logic is used for its well-known speed advantage over static logic and because it guarantees that the outputs are hazard-free. However, a single stage of domino logic can only realize functions that are monotonic. Thus, to implement all functions, some dual-rail inputs and some dual-rail internal signals are sometimes needed. Moreover, the design style uses one-hot encoded outputs to reduce the overhead of completion detection of the evaluation phase of the domino logic. The completion detection of the precharge phase is simply removed with a timing assumption on the precharge signal. The key advantage of this design style is that the domino logic can be optimized to prioritize the computation of instructions depending upon the instruction frequency, potentially leading to dramatic improvements in average-case delays. The circuits, however, can be large and complex, and thus could benefit substantially from supporting CAD tools.

1.3 Technology mapping

The flow of logic synthesis is often divided into two phases of optimization: technology independent and technology dependent. Two-level and multi-level minimization generate optimized Boolean equations in the technology independent phase and technology mapping maps those Boolean equations onto real gates in the technology dependent phase. These gates can be implemented using FPGA or standard-cell libraries. This thesis focuses on the latter technique. Traditional library-based technology mapping for synchronous circuits minimize worst-case delay of the mapped circuit (along with perhaps other metrics, such as area and power) [11, 33, 39, 64, 74, 75]. Those techniques often consist of two explicit steps: decomposition and covering. A Boolean equation is first decomposed into a network which only consist of a set of base functions (e.g. NAND2 and INV). Then, the decomposed network is covered using library gates. The decomposition ensures that each node of the network can be covered by at least one library gate. It also increases the granularity of the network, which provides more mapping options, yielding better final circuits [64].

For asynchronous circuits, however, those explicit technology mapping steps are seldomly considered. Many asynchronous design methodologies rely on the existence of specialized cell libraries instead of standard-cell libraries [23, 41, 69, 78]. Moreover, the few existing asynchronous technology mappers only focus on ensuring hazard-freedom [37, 50, 67, 68] without the consideration of optimizing the average-case performance.

1.4 Average-case technology mapping

Much of the motivation for developing a technology mapper which optimizes average-case performance arises from the RAPPID (Revolving Asynchronous Pentium® Processor Instruction Decoder) project at Intel Corporation [62]. Burst-mode circuits and one-hot domino circuits are two types of asynchronous control methodologies used in this project. For burst-mode design, two-level hazard-free unmapped sum-of-product (SOP) networks are generated using 3D synthesis [88, 89]. For one-hot domino design, a traditional logic synthesis tool, similar to *espresso*, was used to generate two-level unmapped SOP networks which need not to be hazard-free. The hazard-freedom of their mapped circuits is automatically ensured because domino logic produces only monotonic transitions [80]. DeMorgan's laws, logic duplication, and dual-rail inputs are further applied to convert an unmapped network into a non-inverting network which is the constraint of domino logic implementation [80, 70, 71, 60]. After non-inverting unmapped networks are generated, optimized and decomposed, technology mapping is performed using standard-cell library gates. In the RAPPID project, however, the lack of mapper forced manual mapping which is both labor-intensive and sub-optimal since back-of-the-envelope techniques are used to select a mapping. This motivated us to develop an average-case technology mapper which can both utilize existing EDA tools using standard-cell libraries and take advantages of the event-driven property of asynchronous circuits by optimizing for average-case performance.

The meaning of average-case performance of an asynchronous component depends on the nature of the component and its environment. In the remainder of this section, we describe the average performance metrics for burst-mode and one-hot domino circuits and overview the related advancements this thesis achieves.

1.4.1 Burst-mode circuits

For burst-mode circuits, there are two parameters often used to characterize the performance of burst-mode circuits: *latency* and *cycle time*. The latency of a state transition is the *maximum* delay from the transitions on primary inputs to the transitions on primary outputs. If the state signals toggle concurrently with outputs or after outputs, the latency is determined by the delay of the combinational logic. If the state signals toggle before the outputs, however, the latency includes the delay through the feedback delay elements. The cycle time of a state transition is the *minimum* delay necessary between the end of the input burst and the beginning of the input burst of the next state transition. Note that the cycle time of a state transition is the sum of its latency and the associated fundamental-mode constraint.

For many applications, the optimization of latency is more important than that of cycle-time. Consider a situation in which controllers are responsible for controlling the sequence of various datapath operations [87]. In these applications, the environmental response time corresponds to the delay of the datapath units and thus is relatively long. Consequently, the fundamental-mode constraint is typically easily met and the system performance is determined by the sum of the datapath and controller latencies. Thus, in these applications, the burst-mode cycle time does not adversely affect system performance.

In control-dominated applications, on the other hand, the environmental response time of a controller may just be the latency through another controller and thus can be small. In those cases, the fundamental-mode constraints are harder to meet, and, in some cases, delays must be added to increase the environmental response time. Ideally, these delays should be placed inside the communicating controller such that they delay only the response time of those state transitions that violate their fundamental-mode constraints. Otherwise, it may be necessary to add the delays to the output signals to meet the worst-case fundamental-mode constraint, which unfortunately slows down the environment's response time *uniformly* for every state transition. In this thesis, we assume that the environment is ideal and thus our goal is to minimize the controller's average cycle time. Extensions to minimize the worst-case fundamental-mode constraint are straightforward but not discussed here.

Because an output signal is often specified to change in multiple state transitions, its logic may contain different paths that are excited in different state transitions. To optimize for average performance, the delay along these paths should be prioritized according to the relative frequencies of the associated state transitions. In order to achieve this, we assume that conditional probabilities of all state transitions are either provided by the user or estimated through a behavioral simulation of the circuit in its environment. In the SCSI-INIT-SEND circuit, the conditional probabilities depend on the packet size of SCSI transfers. For example, using a reasonable packet size of 1K bytes, the conditional probabilities from state 3 to state 4 and to state 6 would be 0.999 and 0.001 respectively. Our mapper pre-processes these conditional probabilities, using a Markov chain analysis, to obtain the relative priority of each state transition.

Our technology mapper maps only the combinational logic portion of the controller. However, the mapping choices, and thus the structure of the combinational logic, affect the amount of feedback delays required and the fundamental-mode constraint. To account for this, we use a combination of techniques used in [10] and event-driven simulation to estimate the required feedback delay and fundamental-mode constraint during the mapping procedure. Therefore, our mapper can choose solutions based on not only the latency but also the cycle time.

Consequently, our mapper can either optimize burst-mode circuits for *average latency* or *average cycle time*. The average latency is defined as the weighted sum of all latencies, each of which is weighted by the corresponding state transition probability. The average cycle time is defined as the weighted sum of all cycle times, each of which is weighted by the corresponding probability of the pair of the state and next state transition. The choice is user-specified.

Our mapper performs two steps: decomposition and covering. In the decomposition step, optimized circuit equations are decomposed into a set of available base functions, such as 2-input NAND's and inverters. This decomposition is necessary to ensure that every node of the network can be implemented by at least one library gate. It also increases the granularity of the network, which provides more mapping options, yielding better final circuits [64]. A unique feature of our decomposition algorithm is that it incorporates an efficient heuristic that optimizes the decomposition for average performance. In the covering step, the optimized decomposed network is covered with available library gates with the objective of optimizing average performance under a given area constraint.

Our covering algorithm is motivated by Chaudhary and Pedram's covering algorithm [11] which evaluates area-delay tradeoffs based on static (pessimistic) timing analysis. Novel aspects of our algorithm are as follows.

- Generation of a multi-dimensional area-performance tradeoff surface that extends the notion of a two-dimensional area-delay tradeoff curve proposed in [11].

- Adaptation of an input-pattern-dependent method to consider delays of only *true* critical paths associated with each state transition [35], thereby avoiding the *false path problem*.

- Inclusion of a new heuristic to remove likely-non-optimal mappings from consideration to reduce both CPU run-times and memory usage.

We tested our algorithm on a large set of benchmark circuits. To evaluate the benefit of our algorithm, we compared the average performance of our mapped circuits to the average performance of comparable circuits mapped by an algorithm that minimizes the worst-case delay. The results indicate that our circuits have significantly better average performance and are, on average, significantly smaller, with comparable CPU run-times.

1.4.2 One-hot domino circuits

In this thesis, one-hot domino circuits are specified with a set of *incompletely-specified input patterns*, each associated with a probability that reflects the input pattern's relative frequency of occurrence. In practice, these probabilities can be derived from architectural simulation of the design on typical data. Since only one primary output will evaluate in the circuit, we define *pattern delay*, denoted by *p-delay*(p), as the *pattern arrival time* (the time when an input pattern of primary inputs arrives) at the primary output that evaluates for the input pattern p (incompletely or completely specified). Our objective for the one-hot domino circuit is to optimize *average-case delay* which is the weighted sum of all pattern delays, each of which is weighted by its corresponding probability.

The key obstacle to technology mapping of these circuits is that the pattern delay of a circuit for an incompletely-specified pattern cannot be precisely determined because the critical path is unknown when a primary input is specified to be an "X". In fact, an incompletely-specified pattern can be viewed as a set of completely-specified patterns P_c. All we need to do is to find a new delay function which returns the same delay value or bound for all $p \in P_c$.

Fortunately, one-hot domino circuits have a special property that allows us to easily *bound* the delay for an incompletely-specified pattern. In particular, for each incompletely-specified pattern c, we identify two *representative*, completely-specified patterns, c_l and c_u, that yield lower and upper bounds of the delay for pattern c.

Based on this theory, we propose to reduce the technology mapping problem of one-hot domino circuits to the completely-specified input-pattern dependent approach that we used for the burst-mode circuits. Specifically, we replace each incompletely-specified pattern by one of its two representative patterns. Then, we call our mapping routines which is slightly modified to handle domino logic to minimize the average-case delay. Finally, we use the representative patterns to derive bounds of the average-case delay of the mapped circuit.

We demonstrate our approach with a case study of an asynchronous instruction length decoder (AILD) for Pentium® processors in Intel RAPPID project [62]. In particular, we describe *two* combinational blocks for length decoding which are key components of a fast asynchronous length decoder. Our experiments support three important results:

- The range of average-case circuit delays that we derived by our *representative* patterns is narrow (within 11%), thereby illustrating the precision of our bounds.
- The average-case delays of our results are significantly smaller than the average-case delays of the comparable circuits derived using synchronous techniques, thereby illustrating the potential power of our new technology mapper.
- The average-case delays of our results are dramatically smaller than the worst-case delay of the comparable synchronous circuits, demonstrating the potential performance benefit of asynchronous circuits.

1.5 Thesis organization

This thesis is organized as follows. Chapter 2 reviews the background of technology mapping, burst-mode circuits, one-hot domino circuits, and worst-case timing analysis. Chapter 3 presents the average-case technology mapping for burst-mode circuits. Chapter 4 extends the average-case mapping technique to one-hot domino circuits. Chapter 5 presents our conclusion and future work.

Chapter 2

Background

2.1 Previous work on technology mapping

2.1.1 Technology mapping of synchronous circuits

For synchronous circuits, library-based technology mapping is often reduced to a directed acyclic graph (DAG) covering which can be efficiently approximated by a sequence of optimal tree coverings [33, 64]. The optimized equations (obtained from the technology-independent optimization) are decomposed into a DAG where each node is a base function. Particularly, the DAG is called a *NAND-decomposed* graph if the set of base functions consists of only a NAND2 and an INVERTER [11]. The technology mapping problem is to find a minimum cost (area, delay, or power) of covering of the decomposed graph using available library gates. Keutzer first developed the DAG covering algorithm to minimize area using dynamic programming based approach [33]. Rudell then extended the algorithm to minimize the worst-case delay [64]. Touati et al. implemented Rudell's solution for the unknown load problem by computing the best mapping for each possible output load in bottom-up covering. They also minimized the area under delay constraints by computing the best mapping for each possible required time [74]. Chaudhary and Pedram introduced the concept of building area-delay tradeoff points and removing inferior points for the delay-constrained area-minimized problem. They also re-compute the output load for mapped fanin gates to correct the load offset caused by guessing output load when mapping fanins [11]. In addition, Mailhot et al. use BDD Boolean matching instead of graph-based tree-pattern matching in the covering procedure for first optimizing area and then iteratively improving delay [39]. However, in all cases, they employ pessimistic static timing analysis to determine the worst-case critical paths during the mapping and thus suffer from the false path problem.

For emerging low power systems, power minimization has become a new objective for technology mapping. Tiwari et al. extend Keutzer's DAG covering to minimize power [72]. Tsui et al. exploit decomposition for minimizing average switching activity and extend Chaudhury's delay-area curve to the delay-power curve for delay-constrained power-optimized covering [75]. Zhou and Wong develop an exact decomposition algorithm to minimize the switching activity of a decomposed network [92].

With the advent of deep-submicron technology, incorporating layout optimization with technology mapping has become a new goal. Pedram and Bhat extend the DAG covering to minimize post-layout area or delay [58]. Iman and Pedram apply fuzzy logic to compute fuzzy wire delay and build fuzzy-delay curves in the covering step to minimize area under the delay constraint [31]. Lou et al. use area-delay-CCF (cut cost function) tradeoff curves in the covering step to minimize the area or delay of the linear placement of a mapped circuit [38].

For domino logic, Zhao and Sapatnekar propose a parameterized library mapping algorithm based on dynamic programming in bottom-up fashion to determine the optimal domino configuration which minimizes the area [91]. Puri et al. develop a technique of output phase assignment to reduce the logic duplication in domino logic synthesis [60]. Thorp et al. present several techniques to synthesize domino logic using complex gates [70, 71]. None of these techniques address the issue of optimizing their average-case performance.

The general approach of LUT-based FPGA technology mapping is slightly different from that of the library-based technology mapping. A FPGA chip basically consists of a set of K-input LUTs (lookup-table), each of which can realize any Boolean function of up to K variables. The problem is to transform a general Boolean network into a functionally equivalent K-LUT network. Normally, the mapping step is first to decompose a Boolean network into a network of 2-input gates, and then pack the gates into groups, each of which can be implemented in a LUT. MIS-pga, by Murgai et al., uses Roth-Karp decomposition, kernel extraction, and other logic synthesis techniques to minimize area (the number of LUT) or delay (the depth of the circuit) [47, 48]. Chortle, Chortle-crf, and Chortle-d, by Francis et al., minimize area or delay using dynamic programming and bin-packing techniques [21, 22]. FlowMap, by Cong et al., optimally minimizes the depth in polynomial time based on network flow computation [16]. Rmap, by Schlag et al., maximizes LUT routability [65]. Bhat et al. use a simulated annealing based algorithm to iteratively execute mapping and placement to improve the LUT routability [8]. Chen et al. optimize the LUT mapping and the LUT placement simultaneously to improve the routing wire length [12]. Maple, by Togawa et al., is a simultaneous FPGA mapping, placement, and global routing algorithm which can also minimize the channel congestion [73]. Edge-Map, by Yang et al., optimizes the circuit delay in the re-mapping phase using the general delay model which considers the routing wire delay [83]. The other approach, by Chang et al., first combines decomposed gates into a network of super nodes which represented by ROBDDs, then decomposes the network into LUTs such that the number of LUTs or the depth of the mapped circuit are minimized [20].

2.1.2 Technology mapping of asynchronous circuits

Siegel and De Micheli studied the technology mapping problem for asynchronous circuits that operate in fundamental mode, such as burst-mode circuits. They showed that, with only small modifications, synchronous technology mapping solutions can be *safely* used [68]. More specifically, Siegel and De Micheli use results from Unger [76] to demonstrate that the decomposition into base functions does not introduce hazards. Moreover, Siegel and De Micheli present an algorithm to identify library gates which might in some cases cause a hazard. The results demonstrate that most library gates can be used safely, but some MUXes and AOIs, for example, might cause hazards depending on how they are used. When such gates match to a particular node in the graph, further analysis is used to determine if the match causes a hazard in that instance [68]. The key shortcoming of their work is that the underlying synchronous technology mappers they used are limited to optimizing worst-case performance, not average-case performance. It is this limitation that our algorithms address.

Technology mapping techniques for speed-independent circuits have also been widely investigated. Beerel and Meng develop a synthesis technique to decompose Martin's complex gate implementation style [40] to the circuit which is implemented by simple gates [3]. Siegel et al. study how to further decompose Beerel's simple gates to even simpler gates (e.g. 3-input gates to 2-input gates) without introducing hazards [67]. Burns analyzes the correctness conditions for the decomposition of sequential elements in speed-independent circuits [9]. Kondratyev and Cortadella et al. develop a general technique to decompose and optimize both multi-level logic and sequential elements of speed-independent circuits [34, 17]. Also, Myers et al. investigate the decomposition of timed circuits [50].

2.2 Burst-mode control circuits

This section reviews the specification, implementation, and operation of burst-mode circuits.

2.2.0.1 Extended burst-mode (XBM) specifications

An extended burst-mode (XBM) specification [85, 88] consists of a finite number of states, a set of labeled state transitions connecting pairs of states, and a start state. Fig. 1.2a shows the XBM specification of the SCSI-INIT-SEND circuit, which has a conditional input *cntgt1*, 3 edge inputs (*ok*, *rin*, *fain*), and 2 outputs (*aout*, *frout*). Signals not enclosed in angle brackets and ending with $+$ or $-$ are *terminating edge signals*. For example, in the state transition out of state 0, ok is a terminating edge signal. The signals enclosed in angle brackets are *conditionals* which are *level* signals whose values are sampled when all of the terminating edges associated with them have occurred. For example, in the state transitions out of state 3, $cntgt1$ is a conditional signal. $\langle cntgt1^+ \rangle$ denotes "if $cntgt1$ is high" and $\langle cntgt1^- \rangle$ denotes "if $cntgt1$ is low." A state transition occurs when all of the conditions are met and all the terminating edges have occurred. In the state transitions out of state 3, the circuit behavior is interpreted as: "if *cntgt1* is low when *rin* falls, change the current state from 3 to 6 and lower *aout*; if *cntgt1* is high when *rin* falls, change the current state from 3 to 4, lower *aout* and raise *frout*."

A signal ending with an asterisk (such as rin in state 0) is a *directed don't care* [85, 88]. If a state transition is labeled with a^*, the following state transitions in the specification must be labeled with a^* or with a^+ or a^- (a^+ and a^- terminate the sequence of don't cares, which is the reason these are called terminating edge signals). In addition, a directed don't care may change at most once during a sequence of state transitions it labels, *i.e.,* changes *monotonically*, and, if it does not change during this sequence, it must change in the state transition its terminating edge labels. A terminating edge which is not immediately preceded by a directed don't care is called *compulsory*, since it *must* appear in the state transition it labels [85, 88]. In Fig. 1.2a, ok^+ is a compulsory edge because it must appear in the transition from 0 to 1. rin^+ in the transition from state 5 to 3, on the other hand, is a terminating edge, but not a compulsory edge, because *rin* can rise at any point as the circuit transitions from state 4 through state 5 but it must have risen by the time the circuit enters state 3.

2.2.0.2 3D implementation style

In this thesis, we assume that XBM specifications are implemented as *3D* machines [88]. A 3D machine is formally represented as a 4-tuple $\langle X, Y, Z, \delta \rangle$, where X is a set of primary inputs, Y is a set of primary outputs, Z is a possibly empty set of state variables, and $\delta : X \times Y \times Z \rightarrow Y \times Z$ is a *next-state function*.

The hardware implementation of a 3D machine is a combinational network in which all state variables and some primary outputs are fed back as inputs to the network through delay elements. For example, the 3D implementation of the SCSI-INIT-SEND controller is shown in Fig. 1.2b. Note that the state variables, $s0$ and $s1$, and the primary output $aout$ are fed back as inputs to the network. There are no explicit storage elements such as latches, flip-flops, or C-elements in 3D implementations. [1]

Let the fed-back primary outputs be Y' and the fed-back state variables Z'. Let H represent the set of internal nodes in the circuit. Then the combinational next-state logic of the circuit can be represented by a DAG, $\langle N, E, G \rangle$, where N is a set of nodes $X \cup Y \cup Z \cup Y' \cup Z' \cup H$, and E is a set of edges ($E \subseteq N \times N$), and G is a set of gates that label nodes. Edge $(m, n) \in E$ connects node m to node n. m is called a *fanin* of n and the set of fanins of n is denoted *FI*(n). Gate g associated with node n represents the logic function of that node and is referred to as $g(n)$. If $m \in$ *FI*(n), we say that m is a fanin of gate $g(n)$.

For convenience, the set of nodes in $X \cup Y' \cup Z'$ is referred to as *DAG inputs* (DI's). Similarly, the set of nodes in $Y \cup Z$ is referred to as *DAG outputs* (DO's).

---

[1] *3D-gc machines* which are implemented using generalized C-elements [89] are not considered in this thesis and the technology mapping of these circuits is left as future work.

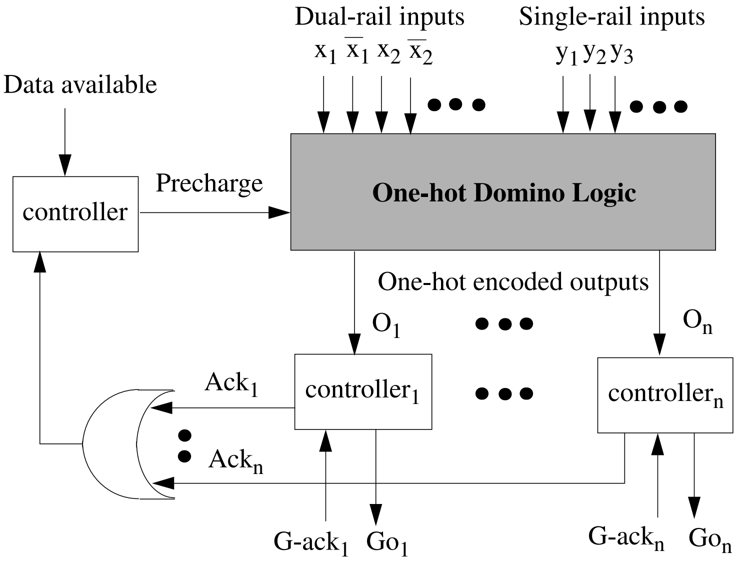

Figure 2.1: A block diagram of the *one-hot domino* logic design style for combinational circuits.

2.2.0.3 State transitions in 3D machines

The state transitions in 3D machines can be designed in two ways: *two-phase* or *three-phase*. In a two-phase state transition, the machine waits for an input burst and then generates a concurrent output and (possibly empty) state burst. In a three-phase state transition, the machine waits for an input burst and then generates a state burst followed by an output burst [85]. Alternatively, three-phase state transitions can generate the output burst before the state burst. To simplify the notation, we focus on the former three-phase design; extensions to handle the latter three-phase design are straightforward.

2.3 One-hot domino circuits

The basic block diagram of a one-hot domino combinational logic block in an environment is shown in Figure 2.1. This section describes the structure and operation of the logic as well as its advantages over other currently known approaches.

2.3.1 The domino core

Domino logic is widely used in high-speed circuits because of its inherent performance advantages. It has smaller parasitic capacitance [80] and separates the pull-up and pull-down events to avoid the fight between the precharge and discharge current [81], often yielding circuits that are faster than circuits obtainable with static CMOS.

Domino logic consists of two types of gates: static CMOS and dynamic precharged gates, both of which must be inverting. As illustrated in Figure 2.2, the type of gates alternates along any path from inputs to outputs. This is sometimes referred to as the *domino constraint*. Notice that we allow the static gate to be *any* inverting CMOS gate [81], whereas, traditionally, the static gate is restricted to be an inverter [80].

Notice that all dynamic (static) gates precharge (discharge) simultaneously during the precharge phase. Thus, the precharge time is fast, and essentially data-independent. Consequently, we need only optimize the evaluation delay of the circuit.

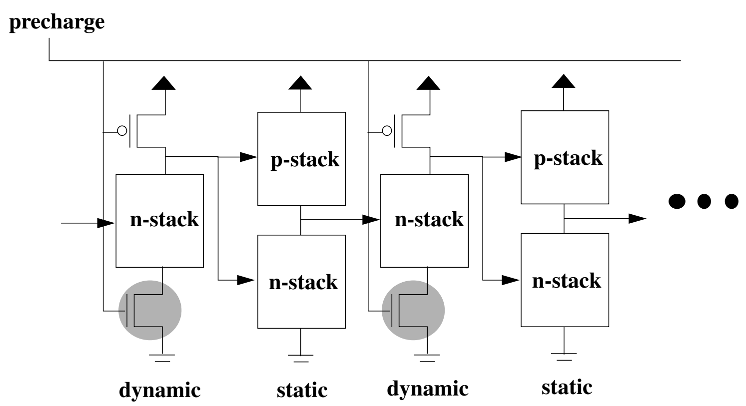

Figure 2.2: An illustration of domino logic.

The gates closest to the primary inputs, referred to as *PI gates*, should be dynamic rather than static. This is because the primary inputs can be assumed to be stable but it is not known whether they will be stable 1 or stable 0 at the start of evaluation or precharge. Consequently, if the PI gate is static, a stable 0 at the primary input may cause a value of 1 to appear at the input to the subsequent dynamic gate at the start of evaluation, possibly causing accidental discharge.

Although we restrict our mapped circuits to the style of domino logic depicted in Figure 2.2, we note that it may be desirable to further optimize the circuits after technology mapping. For example, in some cases, the pull-down transistor driven by the precharge line (shaded in Figure 2.2) can be removed creating what is sometimes called *semi-controlled* domino logic. This can lead to faster evaluation times because it reduces the stack size, but may lead to significant short-circuit current during precharge [81]. The short-circuit current sometimes creates reliability problems, which can be avoided by *staggering* the precharge signal [87].

In addition, we note that charge-sharing problems are always crucial to domino circuits [80]. We assume that either the problems are minimized by precharging each transistor of the pull-down network of each dynamic gate or that further charge-sharing analysis is applied to the mapped circuits.

A key feature of domino logic is that, when designed properly, it can have only monotonic transitions [80]. Consequently, by its very nature, it is hazard-free. It can therefore be easily used in asynchronous circuits by controlling the precharge signal via an asynchronous controller rather than a global clock [27, 87, 82]. Further descriptions of the expected operation of this controller will be given below.

One complication of domino logic is that one stage of domino logic can implement only those functions which are monotonic in their inputs. In particular, binate functions cannot be implemented. Fortunately, this is not a serious limitation because by introducing some dual-rail primary inputs, any function can be implemented [80].

2.3.2 Completion sensing

A naive means of detecting completion of a one-hot encoded combinational logic block is to explicitly derive a done signal from the logical OR of all one-hot encoded outputs. When the done signal rises, the subsequent operation can then be initiated. This means that the start of the next operation is delayed by at least the delay associated with a possibly wide OR gate. Fortunately, there are many instances in which a much better approach can be used.

Consider, for example, the case in which each output O_i should initiate a different operation i, as depicted in Figure 2.1. To implement this, a different controller associated with each operation can be used. When the i-th controller senses signal O_i rising, it can trigger the start of the next operation (by rising Go_i) simultaneously with acknowledging the completion of the one-hot logic (by rising Ack_i). The logical OR of all acknowledgments, Ack_i, can trigger the precharge phase. Thus, the completion sensing delay of the evaluation phase can be completely hidden. We note that this approach was recently used by Benes et al. in the implementation of a high-speed decompression circuit for embedded processors [4].

2.3.3 Precharge phase

In a purely speed-independent implementation the precharging of the logic block must also have some type of completion detection. In this design style, however, timing assumptions can be used to remove the need for an explicit completion detection mechanism. Specifically, all dynamic gates are simultaneously precharged making the precharge time essentially fixed and data-independent. Consequently, control circuitry can easily be used to guarantee that the precharge signal does not become de-asserted until after all gates have been precharged. If some gates are semi-controlled, however, a delay line may be necessary to model the precharge delay [4]. An efficient technique to combine the delay line with the precharge logic for an asynchronous adder is described in [87].

2.3.4 Comparison to other approaches

We first contrast one-hot domino logic with traditional single-rail, bundled-data approaches in which the output control signals are all latched and the output of the latches, which are guaranteed to be hazard-free, are used to drive controllers. Using one-hot hazard-free outputs completely avoids the latch overhead, including the latch propagation delay and the set-up and hold-times. Moreover, single-rail techniques must minimize the worst-case delay among all outputs for all input combinations. Using one-hot techniques, each output can be independently minimized to prioritize the most frequently occurring input combinations which make it fire. Our experimental results suggest that this flexibility can lead to significant speed advantages. The disadvantage of this technique compared with single-rail approaches is that one-hot logic may be larger, domino logic typically consumes more power, and domino logic often requires careful attention to layout to ensure correct operation.

We note that it is also possible to build these combinational circuits using the speculative completion signaling approaches proposed by Nowick et al. [53, 54]. In this approach, the core logic can be optimized for the common case and side logic can be created to identify when common input data arrives and trigger the done signal to designate that the result is obtained. This approach can lead to some reduction in the average-case delay, but it is unclear how easy it would be to generate the side logic for general functions. The advantage of speculative completion approaches is that they can be applied to static logic, which is simpler to design.

We also note that the concept of using domino circuits in asynchronous designs is not new. For example, Williams demonstrated the power of domino circuits very convincingly in his landmark asynchronous divider [82]. In addition, Yun et al. used it effectively in asynchronous adder and multiplier designs [87].

2.4 Worst-case timing analysis

To put our asynchronous timing analysis techniques into perspective, we first briefly discuss several worst-case delay analysis algorithms, including *static timing analysis*, *static sensitization analysis*, and *dynamic sensitization analysis*. To simplify the exposition, we restrict ourselves to circuits composed of simple gates, such as NAND, NOR, AND, and OR, etc.

For a synchronous circuit, the worst-case delay of a combinational block sets a lower bound on the clock cycle time of the circuit. Traditional *static timing analysis* estimates the worst-case delay by computing the topological delay of the longest path [28]. This estimation is pessimistic since the longest path may not be exercised by any input pattern, i.e., no event can propagate along the path. Such a path is said to be *non-sensitizable* and is often called a *false path* [13, 6, 43, 59].

Many researchers have worked on the path sensitization problem to eliminate false paths. *Static sensitization analysis*, which is based on the well-known D-algorithm [63], ignores the timing of each input signal and consequently may underestimate the delay of the circuit [6]. *Dynamic sensitization analysis*, on the other hand, considers the timing of input signals and thus finds the true critical path but requires a computationally intensive task of analyzing all possible input patterns [13]. More efficient input-pattern-independent methods have been proposed to approximate dynamic sensitization analysis, providing an upper bound on the worst-case delay [43, 59].

To the best of our knowledge, these advanced techniques have not been incorporated into a technology mapping procedure. Instead, most technology mappers use more pessimistic, static timing analysis techniques. Consequently these technology mappers can result in non-optimal circuits because of the false path problem.

It is also important to note that all the above techniques adopt the *floating-mode* assumption; i.e., the initial values on circuit nodes are unknown at the time the input pattern is applied. This is because these techniques target combinational circuits in which the sequence of input patterns applied is assumed to be arbitrary.

Chapter 3

Average-Case Mapping on Burst-Mode Circuits

3.1 Average-case performance of burst-mode circuits

This section formalizes our technology mapping objective functions: average latency and cycle time. It first defines the delay through the combinational blocks and the required feedback delays and fundamental-mode constraints. It then formalizes our definitions for average latency and cycle time using Markovian analysis.

3.1.1 Determining the delay through a combinational logic block

This section describes and extends Kung's technique [35] to compute the delay through the combinational logic blocks of burst-mode circuits. Kung observed that in such circuits the sequence of possible input patterns is well-defined and the internal signals are guaranteed to stabilize to known values before the application of each new input pattern. Consequently, he adopted the *single step transition mode* model rather than of the less accurate floating-mode model mentioned above [35].

For our application, we associate one input pattern for each burst that generates transitions at the DO's. Thus, for both two-phase and three-phase state transition, we create a *cld-pattern* modeling the combinational logic delay of the input burst. Three-phase state transitions, however, have an additional cld-pattern modeling the combinational logic delay of the state variable burst.

The values in the cld-pattern correspond to the values of all DI's immediately after the last terminating transition of the burst arrives at the circuit. For example, in Fig. 1.2a, the input pattern for the input burst $\langle cntgt, rin, fain \rangle$ of state transition $3 \rightarrow 4$ is $\langle$ 1, F, 0 $\rangle$.

fain is 0 because it falls during state transition $2 \to 3$. *cntgt* is 1 because we assume that all conditional signals are stable at their sampled value before the compulsory transitions arrive. Note that all state variables introduced by the synthesis process should also be included in the pattern. Moreover, we assume that the user indicates when directed don't care signals change. This latter assumption can be easily generalized, assuming that the user specifies the probability of the directed don't care firing.

The first step of Kung's delay analysis is to perform an extended 8-valued logic simulation to obtain the logic value at the output of each circuit node for each input pattern [35]. In this logic, 1 and 0 denote *stationary* values, R and F denote *transitional* rising and falling values, S0 and S1 denote *static* 0 and *static* 1 *hazardous* values, and DR and DF denote *dynamic* rising and falling *hazardous* values.

To perform the 8-valued simulation, the DI's associated with an input pattern are first set to either a stationary or transitional value depending on whether the input pattern indicates that they have changed compared to their known initial value. These values are then propagated through the circuit using 8-valued truth tables of all gates until the DO's are reached. For example, Table 3.1 shows the 8-valued truth table of a node associated with a 2-input NAND gate.

NAND	1	0	R	F	S1	S0	DR	DF
1	0	1	F	R	S0	S1	DF	DR
0	1	1	1	1	1	1	1	1
R	F	1	F	S1	DF	S1	DF	S1
F	R	1	S1	R	DR	S1	S1	DR
S1	S0	1	DF	DR	S0	S1	DF	DR
S0	S1	1	S1	S1	S1	S1	S1	S1
DR	DF	1	DF	S1	DF	S1	DF	S1
DF	DR	1	S1	DR	DR	S1	S1	DR

Table 3.1: 8-valued truth table of 2-input NAND gate.

Since the circuit is hazard-free, the DO's must have either stationary or transitional values. Internal signals, however, may have hazardous values. Kung observed that because the DO's are hazard-free under the unbounded gate and wire delay model, the delay at any DO cannot be influenced by the delay of internal nodes with hazards. In addition, he realized that any node with a stationary value cannot affect the delay of any DO. Kung thus argued that the delay analysis for a given pattern i can be performed on a reduced circuit in which all nodes with hazardous values, stationary values, and any value which would be eventually suppressed by a stationary 0 value are removed. In practice, this is implemented by associating each input pattern i with the set of nodes $N(i)$, referred to as *sensitized* nodes for pattern i, which have only transitional values that affect DO's. We thus compute the delay of a cld-pattern for only sensitized nodes, thereby savings both run-time and memory. Because we adopt a fixed-delay model, the delay of a node having a transitional value for a given pattern is fixed and it is called *pattern arrival time* of the node.

To define the pattern arrival time, we first review the notions of controlling and dominance [35]. A sensitized fanin of a simple gate (whose output is also sensitized) is controlling if it has a logic value that can independently determine the timing of the output transitional value. For example, a sensitized fanin of a NAND (NOR) gate with a value F (R) is controlling. A sensitized fanin is non-controlling otherwise. Given a cld-pattern i, let $FC(i, g(n))$ denote the set of controlling fanins of $g(n)$. Similarly, $FNC(i, g(n))$ denotes the set of $g(n)'$s non-controlling fanins. If $n \in N(i)$ and g(n) has at least one controlling fanin, the pattern arrival time of $g(n)$ for i, denoted as $pat(n, g(n), i)$, is computed as follows:

$$pat(n, g(n), i) = \min_{m \in FC(i,g(n))} [pat(m, g(m), i) + \textit{p2p-delay}(n, m, g(n), v(i))] \quad (3.1)$$

where *p2p-delay*$(n, m, g(n), v(i))$ is the pin-to-pin delay of gate $g(n)$ from m to n for $v(i)$, the transitional value at n for i. If $n \in N(i)$ and $g(n)$ has only non-controlling fanins, $pat(n, g(n), i)$ is computed as follows:

$$pat(n, g(n), i) = \max_{m \in FNC(i,g(n))} [pat(m, g(m), i) + \textit{p2p-delay}(n, m, g(n), v(i))] \quad (3.2)$$

For example, consider a 2-input NAND node n, with sensitized fanins a and b. Assume that the pin-to-pin delays from a to n and from b to n are 1 and 2 respectively, and the pattern arrival times at a and b for pattern i are 3 and 4 respectively. If the logic values at a and b are F (transitional falling) for this pattern, then a is the dominant fanin of n because both fanins are non-controlling and $min(1 + 3, 2 + 4)$ is 4.

Using Equation 3.1 and 3.2, we can compute the pattern arrival time of each gate recursively in bottom-up fashion. Note that, according to Kung's analysis, Equation 3.1 and 3.2 are the only two possible cases for computing pattern arrival times at sensitized nodes. Also note that the pattern arrival time for DI's can be set by the user. When processing cld-patterns, however, their default value is 0. Once set, repeated application of the above equations yields the pattern arrival times at the DO's. We note that the maximum pattern arrival time between the DO's is the pattern delay through the combinational logic.

3.1.2 Determining the minimum feedback delay and settling times

In addition to the delay through the combinational logic, the performance of a burst-mode circuit is determined by the minimum required feedback delay and the settling times. This subsection describes how these values are determined.

Consider a circuit in which the combinational logic has been mapped but the feedback delay has not been added. To compute the minimum feedback delay to add, our algorithm first determines which gates may exhibit an essential hazard, referred to as *essential-hazard problem gates* [10]. For each essential-hazard problem gate, our algorithm determines the minimum amount of feedback delay that guarantees that the essential hazard does not occur. It then conservatively take the maximum of these delays as the feedback delay. Once the feedback delay is determined, it can compute the settling time required for a given pair of state transitions. It does this by first determining which gates may exhibit hazards due to a violation of the fundamental-mode assumption, referred to as a *fundamental-mode problem gate*. For each gate, it determines the minimum amount of delay the environment must wait before injecting the new input burst. Our algorithm then conservatively take the maximum of these delays as the required fundamental-mode constraint for this state transition.

To identify the essential-hazard and fundamental-mode problem gates our algorithm uses the techniques described in [10]. The essential-hazard problem gates are found by performing an extended-logic simulation of the circuit with a new pattern, called a *fbd-pattern*, in which the feedback state variables change simultaneously with the input burst. Gates which have hazards for this pattern but do not exhibit hazards during the input and state patterns are the essential-hazard problem gates [10]. Similarly, the fundamental-mode problem gates are found by performing a extended-logic simulation with a new *fmc-pattern*, in which the last burst of the current state transition change simultaneously with the input burst of subsequent state transition. Gates which have hazards for this pattern but do not exhibit hazards during the last burst and next input burst separately are the fundamental-mode problem gates [10].

Our approaches to compute the minimum feedback delay and settling time for a given problem gate are also similar. Consider computing the minimum feedback delay required for a given problem gate g. Our algorithm first derives the *propagation delays* to the fanins of gate g excluding the feedback delays (which at this stage of the design process are unknown). Consider the case where the essential hazard at g is caused by a fanin f_1 changing in response to the changing feedback state variable occurs sooner than a second fanin f_2 changing directly in response to the input burst. The minimum feedback delay necessary to avoid an essential hazard at this gate is obtained by subtracting the propagation delay to f_2 plus the corresponding pin-to-pin delay with the propagation delay to f_1 plus the corresponding pin-to-pin delay.

When calculating the propagation delay to the fanin of problem gates (such as f_1 and f_2 described above), we need to consider the delay of nodes that exhibit hazards. Consequently, the delay calculation used for cld-patterns, which ignores hazards, is not applicable. The basic issue to be resolved is that hazardous nodes may transition multiple times in response to an fbd- or fmc-pattern and the delays of these transitions must be considered. To do this, our algorithm uses two different approaches.

The first approach is to use standard event-driven simulation in which we associate an arrival time with each transition edge of a hazard caused by a fbd- or fmc-pattern. At fanin of a gate, each transition edge representing an input event may cause an output event whose associated arrival time is computed as the arrival time associated with the input event plus the corresponding pin-to-pin delay of the associated gate. For the feedback delay computation described above, the propagation delay of f_2 (f_1) is set to be the arrival time associated with the latest (earliest) event of f_2 (f_1). This ensures that the minimum feedback delay is not underestimated.

One disadvantage of this approach is that an exponential number of arrival times is possible for a single node. Consequently, although computationally feasible in our decomposition routine, as is made more clear in Section 3.3, it becomes too costly to use during our covering routine. Thus, during covering, for each node, our algorithm computes a single arrival time that is an upper bound of the time needed for the node to settle. Specifically, our algorithm computes the latest possible arrival time when the last event (edge) of a hazard can occur. For example, consider a 2-input AND gate driving node n, with fanins a and b that exhibits a static 0-hazard due to a rising and b falling. Because the last edge of the static 0-hazard can occur only after b falls, the latest arrival time of node n is the arrival time of b falling plus the corresponding pin-to-pin delay of the gate. To simplify notation, we refer to this computed arrival time as the pattern arrival time for fbd- and fmc-patterns.

Note that more advanced techniques in which the delay of the hazard is captured using both an upper and lower bound is also possible (e.g., see [10]). However, we have found that using a single upper bound is adequate and has the benefit of reducing time and space requirements significantly.

3.1.3 Formalizing our average performance objective functions

This subsection describes how to obtain the probability for each state transition. Using these probabilities, it then formalizes the two objective functions we optimize for: *average cycle time* and *average latency*.

3.1.3.1 The probabilities of state transitions

First, we model the execution of the circuit with a stochastic process $\{B_n, n = 0, 1, 2, \ldots\}$. B_n (the n^{th} state of the process) equals state transition t if the n^{th} state transition executed by the process is state transition t. Consider the circuit depicted in Fig. 1.2. If from the initial state (state 0) of the circuit, the machine executes state transitions $0 \to 1$ followed by state transitions $1 \to 2$ and $2 \to 3$, then $B_1 = 0 \to 1$, $B_2 = 1 \to 2$ and $B_3 = 2 \to 3$.

Second, we let a *trace* T of the circuit be a sequence of specified state transitions, starting with the initial state. We denote by T^k all possible traces of length k. For each such trace $T \in T^k$ we associate a probability which reflects how common this trace is with respect to other traces of the same length. These probabilities are subject to the constraint that for a given length k the probabilities of the traces of length k add up to 1, *i.e.,* $\sum_{T \in T_k} Pr_{\mathit{XBM}}(T) = 1$.

The long term proportion of a transition t, denoted Π_t is the long term proportion of states that the stochastic process is in transition t:

$$\Pi_t = \lim_{k \to \infty} \frac{\sum_{T \in \mathcal{T}^k} (\text{\# of } t \text{ in } T) \cdot Pr_{\mathit{XBM}}(T)}{k} \tag{3.3}$$

Markov chain theory is then used to obtain the long-term proportion of state transitions. A Markov chain is a stochastic process X_n in which the conditional distribution of X_{n+1} is independent of past states and only depends on the value of B_n. If we assume there is no correlation between subsequent environmental choices, then this condition holds. A Markov chain is irreducible if every state can be reached from every other state. If the XBM specification is strongly-connected, this condition holds. Fortunately, in practice, these assumptions are usually satisfied or represent a reasonable approximation of reality. In addition, for XBM specifications that are not strongly-connected, simple extensions to this analysis apply.

For an irreducible Markov chain it can be shown that the set of Π_t's for each transition t are the unique non-negative solutions to the following set of equations [61]:

$$\Pi_t = \sum_{t' : sink(t') = source(t)} \Pi_{t'} \cdot Pr_{\mathit{XBM}}(t) \tag{3.4}$$

$$\sum_{t \in \delta} \Pi_t = 1 \tag{3.5}$$

where $Pr_{\mathit{XBM}}(t)$ is the conditional probability of state transition t.

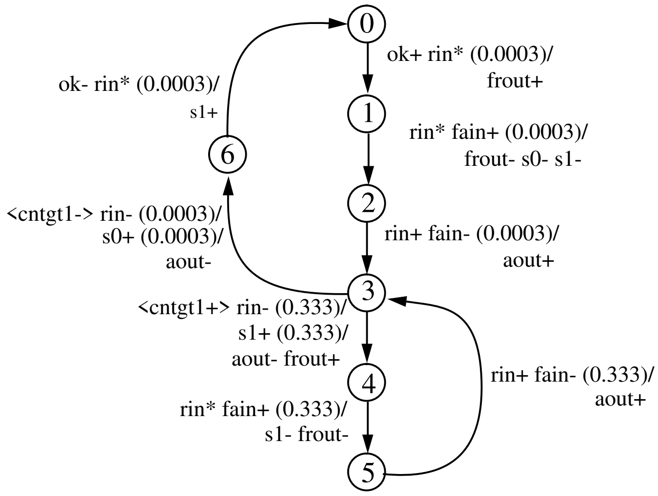

Figure 3.1: Specification after state encoding annotated with results of our stochastic analysis.

Consider the SCSI-INIT-SEND controller described earlier. Given the conditional probabilities discussed in the introduction, these equations show that Π_t for each state transition t has one of two weights. All state transitions in the reset cycle have weight 0.0003 and all state transitions in the main cycle have weight 0.333. To verify these numbers, first, notice that the sum of weights of all state transitions adds up to essentially 1. (The difference is caused by the limited precision of the weights.) Next, consider the equations dictating the weight of state transition $3 \rightarrow 4$

$$\begin{aligned}
\Pi_{3\rightarrow 4} &= \Pi_{5\rightarrow 3} \cdot 0.999 + \Pi_{2\rightarrow 3} \cdot 0.999 \\
&= 0.333 \cdot 0.999 + 0.0003 \cdot 0.999 \\
&= 0.333,
\end{aligned}$$

as expected. The equations dictating the weight of the other state transitions can be similarly verified.

3.1.3.2 Definition of average latency

As mentioned earlier, latency is the critical performance metric of burst-mode circuits in which the environment is slow. Informally speaking, latency is the time after which the input burst has arrived and the output burst is generated and, if not hidden by the concurrent operation of other datapath components (e.g., multiplexing) [87], is the performance overhead associated with sequencing datapath computations.

Each burst-mode state transition has a latency. The latency of two-phase state transitions is simply the maximum pattern arrival time over all outputs for the transition's input burst, i.e.,

$$lat(t) = \max_{o\in Y}(pat(o, g(o), \textit{i-burst}(t))) \tag{3.6}$$

The latency of a three phase state transitions, on the other hand, is the sum of the maximum pattern arrival time over all state variables for the transition's input burst, the delay through the feedback buffers, and the maximum pattern arrival time over all output variables for the transition's state variable burst. Note that for the state variable burst, the pattern arrival time of the fed-back inputs is initialized to 0.

$$\begin{aligned}
lat(t) \quad = \quad & \max_{o\in Z}(pat(s, g(s), \textit{i-burst}(t))) + \textit{fb-delay} + \\
& \max_{o\in Y}(pat(o, g(o), \textit{s-burst}(t)))
\end{aligned} \tag{3.7}$$

The average latency is defined as the sum of the latencies weighted by the relative probability of the state transitions.

3.1.3.3 Definition of average cycle time

As mentioned earlier, the cycle time of the circuit becomes a critical performance metric when the environment is very fast. *Cycle time*, defined over a pair of sequential state transitions (t, t'), is the *minimum* time between the end of the input burst of transition t and the beginning of the input burst of the next state transition t'. It equals the latency of the state transition t plus the required fundamental-mode constraint needed in preparation of t''s input burst. This fundamental-time constraint includes the feedback delay plus the settling time required for the fed-back signals to propagate into the combinational logic deeply enough to ensure that no unexpected hazards occur when t''s input burst arrives. Thus,

$$cyc(t, t') = lat(t) + \textit{fb-delay} + \textit{settling-time}(t, t'). \tag{3.8}$$

The average cycle time is defined as the weighted sum of the cycle times of all state transition pairs. The weight of each state transition pair (t, t') equals the long-term probability of being in state transition t, Π_t, times the conditional probability that the next transition is t' given the current state transition is t.

3.2 Decomposition for average-case performance

The goal of decomposition is to transform a circuit into an equivalent network that consists of only a set of base functions. These base functions form a "fine-grain" representation which provides the subsequent covering step with more flexibility in how to cover the network using library gates. By applying standard tree-decomposition technique [64], for a given circuit, we create a NAND-decomposed network whose base functions are INVERTERs and 2-input NANDs. The results of Unger [76] and Kung [36] ensure that this process preserves hazard-freedom.

An important observation is that there may be many possible NAND-decomposed networks obtainable through tree decomposition but *they are all hazard-free*. The choice of which NAND-decomposed network to use, however, can have a significant impact on the performance of the final mapped circuit. Thus, in order to obtain a better final mapped circuit, we propose an efficient heuristic to find a NAND-decomposed network with optimal performance.

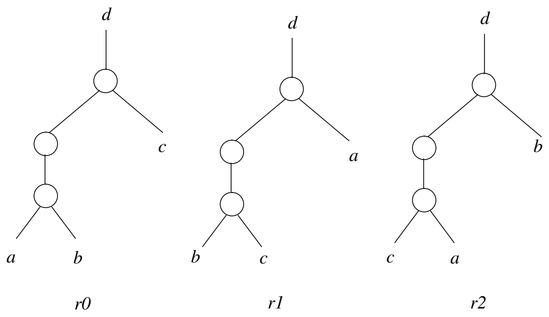

Figure 3.2: The three isomorphic 3-input NANDs.

Our heuristic is motivated by Rudell's well-known optimization for worst-case delay [64]. Rudell repeatedly identified each 3-input NAND subgraph in a decomposed network and rearranged their fanin cones to optimize the worst-case latency of the network. We call this rearrangement of fanin cones "NAND3 rotation." Each NAND3 rotation creates one of three isomorphic NAND3 subgraphs, as illustrated in Fig. 3.2. Consider the isomorphic NAND3 labeled r_0. By reconnecting fanin a to input c, fanin c to input b, and fanin b to input a, we complete a clockwise "rotation" of fanins and produce the isomorphic NAND3 r_1. Rudell repeatedly rotates each NAND3 subgraph in a decomposed network such that the latest arriving input of the NAND3 is pushed closer to the output. Similarly, our heuristic repeatedly rotates each NAND3 subgraph in a decomposed network and chooses the rotation which minimizes the user-specified cost function. Since NAND3 rotation is equivalent to simply choosing a different tree decomposition of the network, it is hazard-preserving. Currently, the user can specify as a cost function either average latency or average cycle time, however, extensions to any combination of the two are trivial.

Although this heuristic is somewhat simple, our experimental results (see Section 3.4) demonstrate that it is both computationally manageable and very effective.

Pseudo-code of our implementation is given in Algorithm 3.2.1. Note that it calls Algorithm 3.2.2 to do rotation on each NAND3 of a network and uses a while loop to repeatedly rotate all NAND3s in the network until no further improvement is possible. More specifically, the objective function we measure is based on two parameters: the average latency

Algorithm 3.2.1 (Rotate decomposed network)
rotate_decomposed_network(network) {
  do
    foreach *nand3* m *in network*
      rotate_nand3(m*)*
  while *(no improvement in performance objective function)*
}

Figure 3.3: Heuristic rotation algorithm for decomposed networks.

or cycle time of the network (depending on user specification), and the average pattern arrival time of the node n where NAND3 m is rooted. The average latency or cycle time of the network is the principle objective function and the average pattern arrival time of the node is used to break ties in the overall average latency/cycle time of the network. This tie breaking feature often allows us to move out of local minimums.

Note also that to recompute the average latency after each NAND3 rotation, the extended logic and event-driven simulation must be re-performed. Moreover, the average cycle time the fundamental-mode constraints must also be re-calculated. To save run-time, our algorithm re-analyzes only those nodes affected by the rotation, i.e., the nodes in the rotated NAND3 and in their transitive fanout. Since the network is unmapped, we assume that the circuit is trivially mapped to NAND2s and inverters and use estimated delays and load capacitances for these gates when simulating.

3.3 Covering for average-case performance

Our covering procedure is divided into two stages. First, a postorder traversal (from DI's to DO's) uses dynamic programming to determine a set of covering solutions rooted at each node. Then a preorder traversal (from DO's to DI's) selects gates which maximizes the user-specified desired performance (latency or cycle time) subject to a given area constraint.

Algorithm 3.2.2 (Rotate NAND3)
rotate_nand3(m) {
  for r = r_1 *to* r_2 /* *rotate* m *clockwise twice* */
    compute new average performance and average pattern
      arrival time for m *rotated to* r
    if *(new average performance is better than previous best)*
        OR
      (
        (the average performance is the same as previous best)
          AND
        (the average pattern arrival time at m *is better than*
          previous best average pattern arrival time at m*)*
)
      rotate m *to* r
      update best average performance and pattern arrival times
}

Figure 3.4: NAND3 rotation algorithm.

3.3.1 Postorder traversal for bottom-up matching

3.3.1.1 Algorithm overview

We first introduce our terminology, some of which is borrowed from [11]. The NAND-decomposed graph is referred to as a *subject graph* and the set of available library gates is denoted by L. A *match* h for a node n in the subject graph is a *pattern graph* of L that is isomorphic to a subgraph rooted at node n and satisfies the conditions found by Siegel and De Micheli to ensure hazard-freedom [68]. Each match h is associated with a set of gates which have the same pattern graph. The set of nodes in a match h for a node n is referred to as $merged(n, h)$. The set of fanin nodes of $merged(n, h)$ is denoted by $inputs(n, h)$.

A *point* p is a tuple $\langle h.g, pat, fbd, st, early, late, area\rangle$ that contains the key parameters of a cover of a subgraph rooted at a node n. First, the point identifies the *matching gate* $h.g$ which is used to derive the point. In addition, the point includes the function $pat : P_i \to \mathcal{R}$ which returns the pattern arrival time, $pat[i]$, for a set of input patterns P_i for which i is sensitized, which as mentioned earlier, can include the following. For a two-phase state transition, there is one *cld-pattern* modeling the combinational logic delay of the input burst, one *fbd-pattern* for determining the feedback delay, and one *fmc-pattern* for each possible next state transition for determining the corresponding fundamental-mode constraints. For a three-phase state transition, there is an additional *cld-pattern* associated with the state variable burst and an additional *fbd-pattern* to determine the feedback delay needed to avoid essential hazards caused by the state variable burst. From these pattern arrival times, our algorithm estimates the minimum feedback delay fbd needed to avoid essential hazards for this mapping of the subgraph rooted at node n. In addition, our algorithm computes the settling times st ($st : \delta \times \delta \rightarrow \mathcal{R}$) that are required for pairs of subsequent state transitions needed to ensure that no fundamental-mode constraint is violated. $early$ and $late$ are bookkeeping vectors used to store sets of fanins associated with the early and late transitions involved in any potential hazard at that node. Finally, $area$ is the area at node n. Since many pattern arrival times are stored in a point, a point is actually multi-dimensional and the set of points forms a multi-dimensional "surface." This is an extension of the two-dimensional curve in [11].

Algorithm 3.3.1 shows the pseudo code for building an area-performance tradeoff surface at a node n. First, the set of matching gates $h.g$ at n are found using the modified matching algorithm developed by Siegel and De Micheli [68]. For each matching gate, the algorithm creates many possible area-performance points depending on how the cones of logic rooted at the input of the match h, $inputs(n, h)$, are mapped. Because the nodes are visited in postorder, the possibly optimal ways to implement the cones of logic rooted at $inputs(n, h)$ have already been analyzed and stored in different points on surfaces at $inputs(n, h)$.

For each combination of surface points at $inputs(n, h)$, an area-performance point is computed as follows. To compute the average cycle time for a point p, the first step is to compute the pattern arrival time for every cld-pattern, fmc-pattern, and fbd-pattern. For each fbd-pattern $i \in P_{fbd}$, the algorithm then selects the maximum feedback delay among the feedback delays stored in in_pts as the default feedback delay for p for resolving essential hazards which may occur in the fanin cones of h. If g is an essential-hazard problem gate rooted at n for pattern i, the algorithm also estimates a new feedback delay to avoid an essential hazard occurred at gate g when i is applied (as is explained in Section 3.3.1.2). If the default feedback delay is smaller than the new computed feedback delay, the feedback delay for p is updated using the new feedback delay. For each fmc-pattern $i \in P_{fmc}$, the fmc for p is estimated similarly. But instead of only one fbd being stored in p, an array of fmc's for all fmc-patterns is stored. The area of the point is the area of $h.g$ plus the total area of in_pts. Then, the point is inserted into the surface and the function *remove_inferior_pts()* removes all surface points which are deemed unlikely to lead to an optimal covering (as is described in Section 3.3.1.5).

Algorithm 3.3.1 (Compute area-performance surface)

```
compute_area-performance_surface(n) {
  foreach match h rooted at node n
    find all combinations of surface points from the surfaces
      of inputs(n, h)
    foreach matching gate h.g
      foreach point combination in_pts
        p = allocate_point()
        foreach i ∈ P_cld + P_fbd + P_fmc
          p.pat[i] = comp_pat_arv_time(n, h, i, in_pts)
          if i ∈ P_cld then apat += w[i] * p.pat[i]
        p.fbd = select_max_fbd_at_in_pts(in_pts)
        foreach i ∈ P_fbd
          if problem_cover(n, h, i) = YES
            fbd = estimate_max_diff(n, h, i, in_pts)
            p.fbd = max(fbd, p.fbd)
            p.early[i] = fanins with early transitions for i in p
            p.late[i] = fanins with late transitions for i in p
        foreach i ∈ P_fmc
          p.st[i] = select_max_diff_at_in_pts(i, in_pts)
          if problem_cover(n, h, i) = YES
            st = estimate_max_diff(n, h, i, in_pts)
            p.st[i] = max(st, p.st[i])
            p.early[i] = fanins with early transitions for i in p
            p.late[i] = fanins with late transitions for i in p
        p.area = Σ_{k=0}^{l} (in_pts[k].area + m.area)
        insert p into n.surface in area-ascending order
        remove_inferior_pts(p, n.surface)
}
```

Figure 3.5: Computing the area-performance surface at a node.

Consider the example depicted in Fig. 3.6 where a matched 2-input AND gate $h.g$ rooted at node D covers nodes C and D. The set $inputs(D, h)$ is $\{A,\ B\}$. For simplicity, we assume that we are optimizing for latency and that there are two equally weighted cld-patterns, pattern 1 and 2, and no fbd-patterns. The transitional values for the two patterns

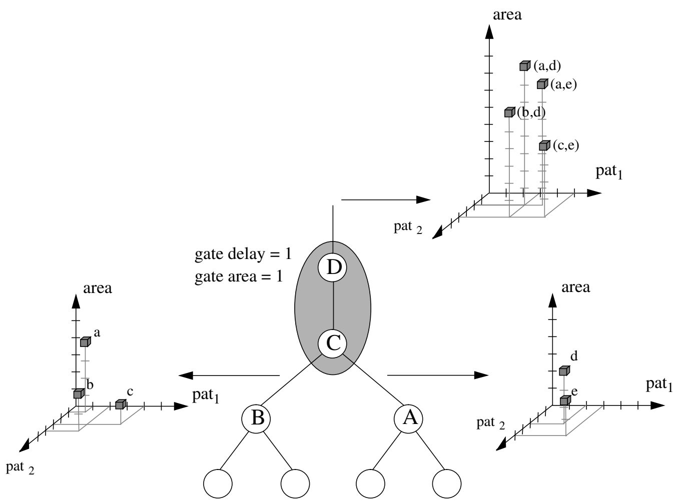

Figure 3.6: Computing the surface points for a match.

at A, B, C, and D are (R,F), (R,F), (F,R), and (R,F), respectively. A has 2 points d, e, and B has 3 points a, b, c. There are 6 possible input point combinations, each of which represents a different input gate combination rooted at A and B and determines one point at D. For example, consider the pair (a, e) and assume that all pin-to-pin delays are 1. Then, $pat[1]$ is $max(1+1, 3+1)$ which is 4 and $pat[2]$ is $min(1+1, 5+1)$ which is 2. The area is the sum of the areas of a and e plus the area of the AND gate which is assumed to be 1, yielding 7. As is explained in Section 3.3.1.5, only four of the 6 possible points generated are deemed likely to lead to a good covering; the other two are removed by the function *remove_inferior_pts()*.

3.3.1.2 Estimating the required feedback delay and settling-times

Recall that the required feedback delay and settling-time of a particular problem gate is obtained from the difference between the propagation delay through the feedback signals and the propagation delay from the primary inputs to the problem gate [10]. Once the combinational logic portion of the circuit is mapped, this value can be readily obtained (as described in Section 3.1.2.) During the covering process, however, the delay through the feedback signals is not known since the nodes are processed in postorder. In particular, the logic between the problem gate and the corresponding state variables/primary outputs has not yet been chosen.

To address this problem, our algorithm estimates the delay of each state variable/primary output z to be the pattern arrival time of z in the un-mapped network. Although this estimate is somewhat coarse, our experimental results suggest it is effective.

3.3.1.3 Extending pattern arrival times to complex gates

To calculate the pattern arrival time during covering, function *comp_pat_arv_time()* must analyze matching gates that may be complex. For such gates, we algorithmically extend our definition of a dominant fanin. Specifically, we recursively traverse the NAND-decomposed subgraph representing the complex gate to simultaneously find the dominant fanin and pattern arrival times for a given point p. For each complex-gate input m, we calculate $p.pat[i]$ + *p2p-delay*$(m, n, h.g, t)$, where t is the transition that node n makes in pattern i. We then recursively propagate these delays up through the subgraph selecting the delay and fanin node which is dominant for the current subgraph node. For example, consider a matching AOI gate $h.g$ rooted at node n where all fanins are non-controlling. The algorithm first chooses the maximum sum in each AND subgraph and then chooses the minimum sum for the OR subgraph among those maximum sums of AND subgraphs. The corresponding m for the minimum of the maximums is the dominant fanin m_d of n and the minimum of maximums is the pattern arrival time $p.pat[i]$.

3.3.1.4 Accounting for unknown loads

The function *comp_pat_arv_time()* uses the pin-dependent delay model in MIS [11] to model the delay of $h.g$ from input pin m to output pin n, as follows,

$$\textit{p2p-delay}(n, m, h.g, t) = \tau(n, m, h.g, t) + R(n, m, h.g, t) \cdot C_n \tag{3.9}$$

where $\tau(n, m, h.g, t)$ is an intrinsic pin-to-pin delay of the gate, $R(n, m, h.g, t)$ is a pin-specific output drive-resistance, and C_n is the capacitive load at the output of $h.g$.

It is important to note that when the postorder procedure first visits each $m \in inputs(n, h)$, the exact output load C_m is unknown because some of its fanouts may not have been mapped. In particular, this node may have fanouts in cones of logic that have been traversed as well as cones of logic that have yet to be traversed. For this reason, our algorithm must estimate its load. For the mapped fanouts it accounts for the input capacitance of the mapped gate. For the unmapped fanouts it uses a default input capacitance C_d. This is known as the *unknown load approximation.*

A useful observation made in [11] is that when the main procedure visits node n and computes a point for $h.g$, the exact output load for each node m is more precisely known because the input capacitance of its fanout n is determined by the gate $h.g$. More specifically, the previously unknown load contribution of $h.g$ is now known.

To obtain a better estimate of the pattern arrival time at n, the routine *comp_pat_arv_time()* compensates for the difference between the default load and the exact load by adding a *load shift* term to the pattern arrival times at m. Specifically, this *load shift* term is $R(m, m_d, h.g', t') \cdot (C_m - C_d)$, where m_d is the dominant fanin of m and t' is the transition for pattern i at node m [11].

3.3.1.5 Removing points with inferior average performance

The criteria of determining the inferiority of a point affects the efficiency and effectiveness of the algorithm. A good criteria reduces a large number of points but still keeps the final solution as close to optimal as possible.

We first propose an exact technique to remove inferior points that depends on what the tool is optimizing. When optimizing for latency, a point is *inferior* if there exists another point on the surface which has less or equal latency for *all* state transitions. When optimizing for cycle time, the other point must have less or equal to cycle times for *all* state transition pairs. These definitions yields the optimal covers subject to load shift errors. However, under this definition, it is difficult for a point to be inferior since any individual latency/cycle time may violate the above condition. Our experimental results in Section 3.4 show that the number of non-inferior points becomes unmanageable for large circuits.

Thus, in order to reduce the number of points, we propose a new heuristic to quantify the potential merits of one point over another based on the notions of a point's average latency and cycle time. Specifically, the *point average latency* is an estimate of the average latency of the circuit up to that node for the cover of the subgraph represented by the point. It is defined as the sum of the point's state transition latencies weighted by the probability of the state transition. For a two-phase state transition the point's state transition latency is simply the point's pattern arrival time for the cld-pattern associated with this state transition. For a three-phase state transition, the point's state transition latency also consists of the pattern arrival time for the cld-pattern associated with the state burst and the current estimated value of the required feedback delay.

Similarly, the *point average cycle time* is an estimate of the average cycle time of the circuit up to that node for the cover of the subgraph represented by the point. It is defined as the point average latency plus the current estimate of the average required fundamental-mode constraint.

A point has *inferior average performance* if there exists any other point on the surface which has less or equal area and less or equal average performance, i.e., latency or cycle time depending on which the user directs the mapper to optimize. Our experimental results indicate that judging inferiority based on point average latencies and cycle times is very effective. In other words, these quantities adequately reflect the relative overall performance impact of various mapping options for a given subgraph.

In our implementation, we maintain the area-performance surface as an area-ascending ordered list to make removing average performance inferior points efficient. Specifically, a point p divides the list into two sets A and B. Each point in set A has no larger area than the area of p and each point in set B has larger area than p. If there exists any point in set A with better (or equal) average performance than p, p has inferior average performance and is removed from the list. If not, p is not removed and any point p' in set B that has worse (or equal) average performance than p is removed. Thus, the list remains not only area-ascending order but also average-performance-descending order by construction. Consequently, removing inferior average performance points takes linear time with respect to the number of points in the list.

Recall that in Fig. 3.6 only 4 points are stored at D instead of 6 since both (c, d) and (b, e) generate likely inferior points. This is because (b, e), (c, d), and (c, e) all have the same average latency of 4.5 but point (c, e) has the lowest area of 4.

It is important to note why dropping points with inferior average performance may lead to a sub-optimal solution. Consider the case where the critical path for Pattern 2 is not through this AND gate and is through a different part of the graph. Then, the only pattern arrival time of importance is Pattern 1. For the point from (b, e), the pattern arrival time for pattern 1 is 4. Because this is the smallest pattern arrival time for Pattern 1 between the points with an area of 4 or higher, it could be part of the optimum solution and should not be dropped. Fortunately, our experimental results indicate that this heuristic effectively removes most surface points yet still maintains close to optimal solutions. This is because if the heuristic picks a point that yields a non-optimal solution, the difference from the optimal solution is probably small since the point would most likely have significantly sub-optimal pattern arrival times for only very infrequent patterns.

We also note that for multi-fanout nodes, we adopt Chaudhary and Pedram's heuristic of dividing the area of each point by the number of its fanouts [11]. This means that the relative area cost of solutions in which the multi-fanout node is internal to a gate is very high. Consequently, this heuristic favors solutions in which the multi-fanout node is a gate output rather than internal to a gate.

In addition, to further reduce the number of points, we adopt the well-known binning technique [64] in which if the difference between two performance metrics is less than a user-defined *bin size*, the performance metrics are considered equal, yielding many more inferior points.

3.3.2 Decomposing the mapping of multi-output circuits

Ideally, once area-performance tradeoff surfaces at all DO's are computed, we would select a combination of DO points (one for each DO) which yield the best average performance over all possible point combinations that satisfy a given total area constraint. This, however, can be computationally very expensive since the number of DO point combinations that would need to be analyzed can be huge. Thus, we propose a fast heuristic to find a good, but potentially non-optimal, solution which is to iteratively cover each output cone individually and sequentially, using a decomposed area constraint. Another important advantage of this approach is that once an output cone is covered, all surface points on nodes in that cone can be freed, dramatically reducing the total memory needed for the algorithm.

It may be useful to explain why this heuristic may not yield a circuit with optimum performance. Consider the case where we are optimizing the average latency of a circuit that has two output cones, referred to as A and B, and two state transitions, t_1 with probability 0.9 and t_2 with probability 0.1. Assume for simplicity that these state transitions have only two-phases and thus feedback delays need not be considered. In the frequent state transition t_1 assume both outputs transition, whereas in the less frequent state transition t_2 assume only the output of cone A transitions. Because we process cones individually and sequentially, when we optimize output cone A we favor the pattern arrival times corresponding to t_1. Unfortunately, it might be the case that the latency of t_1 is dictated by output cone B. Thus, in this case, the ideal thing to do would be to optimize A for the less frequent state transition t_2 (rather than t_1) because this would yield lower average latency.

This problem occurs because cone A is only critical for relatively infrequent patterns. Fortunately, this means that the difference from the optimal solution is usually small since only the pattern arrival times of relatively infrequent patterns remain non-optimal. In addition, this problem can be mitigated by relaxing the required times for state transitions at a DO to be equal to the maximum corresponding pattern arrival times obtained from previously mapped cones. Also note that our estimate of the required feedback delay and settling times are automatically updated during the process of mapping each cone. Thus, cones mapped later in the algorithms have the benefit of more accurate estimates.

3.3.3 Preorder traversal for top-down selection

Now assume we have selected a point in a DO's surface whose performance is the best of all points that satisfy the given local area constraint. Since the area and performance of this point is based on the unknown load approximation, the set of gates that lead to this point may actually yield different area and performance characteristics from that estimated by the point [11]. Thus, our algorithm performs a separate preorder traversal to actually select the remaining gates in the cover of this cone. In this way, we can keep the overall difference between estimated and actual area and performance metrics to be typically less than 10%.

The first step of Algorithm 3.3.2 is to call *find_best_point()* to find the best point at node n. To facilitate this, the node data structure contains an array of *pattern required times*, a corresponding array of *directions of importance*, an array of *settling required times*, and a *feedback required delay*. The direction of importance for a given pattern indicates whether the pattern arrival time of the point chosen should have a larger or smaller value. The direction of all cld-patterns is always smaller because the chosen point should have a delay equal to or smaller than the required times. However, for fmc-patterns and fbd-patterns, both directions are needed. Recall that our algorithm computes the settling time and feedback delay by taking the difference of two pattern arrival times, one corresponding to an early transition and one corresponding to a late transition (these are computed from $req\_max\_fbd.early[i]$

and $req\_max\_fbd.late[i]$ respectively). To reflect the desire for the point to have a smaller difference, the chosen point's pattern arrival time for the early transition needs to be equal to or *smaller* than the required pattern arrival time while the point's pattern arrival time for the late transition needs to be equal to or *larger* than the required pattern arrival time.

Due to the unknown load problem, it may not be possible to find a point that satisfies all these constraints. For this reason, as illustrated in Algorithm 3.3.3, we compute a *load shift error* to represent by how much a point violates these constraints. Specifically, the load shift error is the weighted sum of the amounts that the pattern arrival time violates the corresponding pattern required time. For a point to satisfy all pattern required times, the load shift error must equal zero. When no point satisfies all the pattern required times, the function selects the point with the smallest load shift error.

After finding the best point, Algorithm 3.3.2, sets the pattern required times for all fanins of the chosen gate. Note that for a DO n, whose point is selected by the user, $prt(n, i) = n.best\_p.pat[i]$ for all patterns i for which n is sensitized and $prt(n, i) = \infty$ for all other patterns. For all other nodes, the algorithm initializes the pattern required times as follows.

$$prt(m, i) = prt(n, i) - \textit{p2p-delay}(n, m, best\_p.h.g, t), \tag{3.10}$$

where $best\_p.h.g$ is the best matched gate rooted at n stored in point $best\_p$.

Next, Algorithm 3.3.2 checks whether the chosen gate is an essential-hazard problem gate. If so, it computes the slack between $best\_p$'s stored feedback delay and the maximum feedback delay allowed. This slack, which can be either positive or negative, is used to compute a secondary pattern required times for the fanins. If this secondary pattern required time is more constraining than the initially computed pattern required time, the pattern required time is updated accordingly. The algorithm performs a similar process if the chosen gate is a fundamental-mode problem gate. Note that we initialize the maximum required feedback delay and settling times at the DO's to the values in the point $best\_p$ selected at the DO.

Algorithm 3.3.2 (Recursively select best gates)

```
recur_select_best_gates(n) {
  n.best_p = find_best_point(n)
  foreach m ∈ inputs(n, n.best_p.h.g)
    foreach i ∈ P_cld
      p2p-delay(m, n, n.best_p.h.g, t(i)) =
        load(n) * n.best_p.h.g.drive[t(i, n)] + n.best_p.h.g.intrinsic[t(i, n)]
      m.prt[i] = n.prt[i] − p2p-delay(m, n, n.best_p.h.g, t(i))
    /* propagate req_max_fbd top-down */
    m.req_max_fbd = n.req_max_fbd
    foreach i ∈ P_fbd
      p2p-delay(m, n, n.best_p.h.g, t(i)) =
        load(n) * n.best_p.h.g.drive[t(i, n)] + n.best_p.h.g.intrinsic[t(i, n)]
      m.prt[i] = n.prt[i] − p2p-delay(m, n, n.best_p.h.g, t(i))
      /* flags for comparing prt[i] and pat[i]
        set to be SMALLER if m is a DO */
      m.g_or_s[i] = n.g_or_s[i]
      if problem_cover(n, n.best_p.h.g, i) = YES
        early = n.best_p.fbd.early
        late = n.best_p.fbd.late
        slack = m.req_max_fbd − n.best_p.fbd
        if m ∈ n.best_p.early[i]
          m.g_or_s[i] = SMALLER
          2nd_prt = early + slack/2 − p2p-delay(m, n, n.best_p.h.g, t(i))
          if m.prt[i] > 2nd_prt then m.prt[i] = 2nd_prt
        else if m ∈ n.best_p.late[i]
          m.g_or_s[i] = GREATER
          2nd_prt = late − slack/2 − p2p-delay(m, n, n.best_p.h.g, t(i))
          if m.prt[i] < 2nd_prt then m.prt[i] = 2nd_prt
    foreach i ∈ P_fmc
      /* propagate req_st[i] top-down */
      m.req_st[i] = n.req_st[i]
      /* the remainder of algorithm is similar to P_fbd section,
        but with every max_fbd and req_max_fbd substituted
        with st[i] and req_st[i] */
    recur_select_best_gates(m)
}
```

Figure 3.7: Algorithm to recursively select best gates for one DO.

Algorithm 3.3.3 (Find best points)

```
find_best_point(n) {
  min_load_shift_error = ∞
  foreach p in n.surface in area-ascending order
    load_shift_error = 0.0
    foreach i ∈ P_cld
      if p.pat[i] > n.prt[i]
        load_shift_error += w[i] * (p.pat[i] − n.prt[i])
    foreach i ∈ P_fbd
      if (p.pat[i] > n.prt[i]) ⊕ (m.g_or_s[i] = GREATER)
        prt_violation =
          max(|p.pat[i] − n.prt[i]|, prt_violation)
    fbd_violation = req_max_fbd − p.fbd
    fbd_load_shift_error = max(prt_violation, fbd_violation)
    load_shift_error += prob(P_fbd) * fbd_load_shift_error
    foreach i ∈ P_fmc
      if (p.pat[i] > n.prt[i]) ⊕ (m.g_or_s[i] = GREATER)
        prt_violation = |p.pat[i] − n.prt[i]|
      st_violation = req_st[i] − p.st[i]
      st_load_shift_error = max(prt_violation, st_violation)
      load_shift_error += w[i] * st_load_shift_error
    if load_shift_error = 0.0 then
      return(p)
    else if load_shift_error < min_load_shift_error
      min_load_shift_error = load_shift_error
      min_load_shift_error_p = p
  return(min_load_shift_error_p)
}
```

Figure 3.8: Algorithm to find the best point in a surface.

3.3.4 Complexity analysis

We focus our complexity analysis on our covering algorithm because it dominates the overall run time. We first consider a match h at node n where the matching gate $h.g$ has l inputs and each input k has $N(k)$ surface points. The number of points at n is bounded by $\prod_{k=0}^{l} N(k)$. Assuming a fixed library size, the total number of matches at a node is a constant. We can thus conclude that the number of points from one level of the graph to the next grows at most polynomially with a degree that is bounded by the maximum number of gate inputs.

The time complexity to derive each point consists of the sum of the time to generate the point and the time to update the node's area-performance surface. The time to generate the point can be assumed to be a constant. The time to insert the point into the surface and remove points with inferior average performance, however, depends on the number of points on the node's surface. If the surface is implemented as a linked list, the time complexity for inserting the $(N(k)+1)^{th}$ point into the surface and removing points with inferior average performance is $O(N(k))$. The preorder traversal also involves a search of the surface of every node k visited, taking $O(N(k))$ time.

Thus, the overall complexity of the algorithm is $O(m \cdot N_{max}^2)$, where m is the total number of nodes in the network and N_{max} is the maximum number of surface points on any surface.

The high computational complexity of the algorithm is mitigated by a variety of factors. First, asynchronous control circuits tend to be relatively small (less than 1K gates). This is because to achieve good performance asynchronous systems are typically designed using a distributed control paradigm consisting of many relatively small controllers running concurrently rather than a single large controller. Second, as mentioned earlier, the number of patterns of interest for each output tends to be small. And, finally, most derived area-performance points have inferior average performance and are therefore dropped. Indeed, our preliminary results, described in Section 3.4, are much more promising than the above worst-case complexity analysis suggests.

3.4 Experimental results

We conducted experiments on a suite of benchmark circuits, presented in Table 3.2, using a AMD-K6-233MHz/Linux PC with 128 Megabytes of memory. Each circuit is specified in an unoptimized Verilog netlist along with an XBM specification annotated with conditional probabilities. Using Markov chain analysis on the annotated XBM specification, we automatically generate the set of input patterns and their associated probabilities. The decomposition and covering algorithms are implemented on top of the POSE environment [30], which is the extension of the SIS framework [66] in which Chaudhary and Pedram's mapper is implemented.

Circuits Tested				
Circuit	# DI's	# DO's	# Pats	# Nodes
merge	6	3	8	58
bufctrl1	5	3	3	66
p3	7	3	11	75
q42	7	3	4	101
scsi-init-send	10	4	10	141
dramc	14	7	14	287
binary-counter	16	7	32	369
pe-send-ifc	15	5	14	385
trcv	21	7	30	746
ircv	21	7	30	779
tsend	34	9	42	1410
p1	45	17	31	1606
isend	45	9	44	2096
scsi	47	11	93	2470
cache-ctrl	54	20	49	3032

Table 3.2: Description of circuits tested.

3.4.1 Decomposition results

We first used the SIS function *tech_decomp* to tree-decompose each benchmark into a hazard-free NAND-decomposed un-optimized network. We then applied our decomposition heuristic to rotate the network optimizing first for latency and secondly for cycle time. Table 3.3

reports the average latency/cycle-time for both the un-optimized and optimized circuits. The run-time for all circuits were less than 1 hour, with all but two circuits taking less than 7 minutes. The performance comparison between un-optimized and optimized circuits demonstrates that a rotated NAND-decomposed network typically has significant improvement (from 4% up to 39%) in average performance compared to its un-optimized NAND-decomposed network.

Experimental Results of Decomposition Heuristic								
Circuit	w/o Rotate		w/ Rotate		Imp.		RunT (s)	
	AveLat	AveCyc	AveLat	AveCyc	AveLat	AveCyc	AveLat	AveCyc
merge	6.79	6.79	4.16	4.16	39%	39%	0.2	0.2
bufctrl1	8.75	8.77	7.96	7.98	9%	9%	0.0	0.1
p3	4.93	4.93	4.59	4.71	7%	4%	0.2	0.2
q42	11.38	11.40	9.80	9.82	14%	14%	0.2	0.2
scsi-init-send	13.62	13.62	10.68	10.68	22%	22%	0.9	0.9
dramc	10.24	10.58	8.52	8.52	17%	19%	4.3	4.3
binary-counter	14.47	14.54	12.31	12.44	15%	14%	11.6	12.7
pe-send-ifc	14.82	14.82	12.29	12.30	17%	17%	10.0	10.3
trcv	25.64	25.64	21.56	21.56	16%	16%	62.9	63.7
ircv	26.04	26.68	20.78	21.12	20%	21%	50.8	52.1
tsend	30.91	31.07	24.20	24.20	22%	22%	399.8	422.9
p1	19.66	20.16	17.22	17.49	12%	13%	219.3	229.1
isend	30.07	30.50	27.46	27.89	9%	9%	406.8	646.5
cache-ctrl	28.64	29.13	23.90	24.23	17%	17%	1857.9	1609.5
scsi	26.35	27.59	20.77	20.71	21%	25%	2561.7	2343.5

Table 3.3: Experimental results of our decomposition heuristic. Delays are measured in nano-seconds.

3.4.2 Covering results

We conducted three mapping experiments on each benchmark: average-case mapping without rotation, average-case mapping with rotation, and worst-case mapping using a modified version of Chaudhary and Pedram's mapper that minimizes worst-case delay.

For all experiments, we used the *lib2* gate library included in the SIS package. For simplicity, we assumed that all gates in *lib2* are hazard-free without running De Micheli and Siegel's algorithm to determine hazardous gates in *lib2*. The matching routine we used is borrowed from SIS and always associates the pin of a simple gate that has the smallest intrinsic delay to the input of a pattern graph with the longest path to the output. This choice of pin association favors covering longer paths of a network with shorter delay, leading to a more balanced covering that effectively minimizes worst-case delay. This unfortunately counteracts our efforts to create an unbalanced covering to shorten true critical paths. In order to truly reflect the potential advantage obtained from the rotation, we reversed the pin delay assignment of simple gates (e.g., NANDs, NORs) such that the pin with the shortest path to the output has the smallest intrinsic delay for our average-case mapping.

3.4.2.1 Evaluation of inferior point schemes

We first present Table 3.4 that shows the number of surface points we allocated and the run-time for each benchmark for both the exact and heuristic point removal schemes. It also lists the average performance using the heuristic and exact technique in the experiments of our mapper with rotation. As we can see, the performance of the circuits obtained with the heuristic and the circuits obtained with the exact technique are almost identical in all cases, demonstrating that the heuristic technique does not significantly reduce the circuits' quality. Moreover, the results demonstrate the memory and run-time efficiency of the heuristic. For example, when optimizing PE-SEND-IFC for average latency, the exact technique with rotation allocated 58,401 points, whereas the heuristic only allocated 1469 points. The run-time of the algorithm is reduced dramatically (1640.8s vs. 17.1s). For the five largest benchmark circuits, we used binning technique to keep the complexity manageable. Given equal bin sizes, the exact technique cannot complete after 1 hour on twelve of the experiments, whereas the heuristic technique completes in all cases.

3.4.2.2 Analysis of impact of rotation on mapped circuits

We next performed experiments to determine the impact on our rotation scheme on the mapped circuits. The results, shown in Table 3.5, show that rotation leads to better mapping in almost every circuit, most often yielding improvements of over 5%.

Interestingly, the results suggest that the improvement in the mapped circuits is sometimes limited by the structure of the circuit. For small shallow networks, such as Q42, the improvement of average latency in the mapped circuits is significantly smaller than

Number of surface points and run-time (sec)														
Circuits	Bin	AveLatency Optimization						AveCycleTime Optimization						
		Heuristic			Exact			Heuristic			Exact			
	Size	AveL	Point	Time	AveL	Point	Time	AveC	Point	Time	AveC	Point	Time	
merge	0.0	1.82	74	0.3	1.82	92	0.3	1.76	75	0.4	1.76	95	0.4	
bufctrl1	0.0	2.14	114	0.1	2.13	156	0.2	2.54	121	0.4	2.62	363	0.5	
p3	0.0	1.01	164	0.5	1.00	602	0.8	1.28	191	1.7	1.23	1925	8.5	
q42	0.0	2.60	217	0.5	2.60	407	0.6	2.77	278	1.7	2.77	1951	5.1	
scsi-init-send	0.0	2.29	293	1.3	2.28	542	1.4	2.29	310	3.1	2.28	1537	7.6	
dramc	0.0	2.08	913	10.2	2.09	6297	58.6	2.16	1001	71.0	2.12	11166	339.3	
binary-counter	0.0	1.54	720	14.8	1.53	2537	17.4	2.95	772	38.4	3.13	20607	313.6	
pe-send-ifc	0.0	2.74	1469	17.1	2.74	58401	1640.8	3.14	1633	90.5	–	–	–	
trcv	0.0	3.57	2820	94.1	–	–	–	3.55	2938	474.8	–	–	–	
ircv	0.0	3.17	2632	76.9	–	–	–	3.18	2623	367.7	–	–	–	
tsend	0.01	3.92	2308	420.5	3.99	10449	1039.3	3.91	2342	615.8	–	–	–	
p1	0.01	3.62	3132	250.7	–	–	–	4.20	3206	668.1	–	–	–	
isend	0.05	4.55	2820	428.0	4.55	6835	757.1	4.62	2818	775.6	–	–	–	
cache-ctrl	0.05	5.34	3818	1900.0	5.39	8774	2283.0	5.44	3789	1851.1	–	–	–	
scsi	0.1	4.21	2566	2605.6	4.29	5542	2722.8	4.39	2588	2540.9	–	–	–	

Table 3.4: Comparison of covering algorithms using exact vs. likely inferiority heuristics. AveL corresponds to average latency and AveC corresponds to average cycle time. In addition, "–" denotes time-out.

observed in the un-mapped circuits. This is because a large reduction of critical path delay can be achieved when covering using unbalanced library graphs without doing any rotation. In some of the larger and deeper networks (e.g., BINARY-COUNTER, TSEND and SCSI), however, rotation yields substantial improvements in the delay of the mapped circuits. This suggests that, sometimes, less gates are needed to cover the critical path of the rotated networks compared to the un-rotated networks.

Average-Case Performance of Mapped Circuits						
Circuits	AveLat			AveCyc		
	ACNR	ACR	Imp.	ACNR	ACR	Imp.
merge	2.01	1.82	9%	1.91	1.76	8%
bufctrl1	2.33	2.14	8%	2.57	2.54	1%
p3	1.04	1.01	3%	1.30	1.28	2%
q42	2.62	2.60	1%	2.86	2.77	3%
scsi-init-send	2.57	2.29	11%	2.68	2.29	15%
dramc	2.15	2.08	3%	2.18	2.16	1%
binary-counter	1.76	1.54	13%	3.28	2.95	10%
pe-send-ifc	3.02	2.74	9%	3.43	3.14	8%
trcv	3.85	3.57	7%	3.95	3.55	10%
ircv	3.38	3.17	6%	3.62	3.18	12%
tsend	4.49	3.92	13%	4.42	3.91	11%
p1	3.95	3.62	8%	4.67	4.20	10%
isend	4.74	4.55	4%	4.76	4.62	3%
cache-ctrl	5.92	5.34	10%	6.13	5.44	11%
scsi	5.13	4.21	18%	5.39	4.39	18%

Table 3.5: Comparison of average performance (measured in nano-seconds) of rotated and unrotated networks. ACNR corresponds to average-case mapping without rotation. ACR corresponds to average-case mapping with rotation.

3.4.3 Comparison to worst-case mapped circuits

We also compare our best average-case mapper (with rotation) to Chaudhary and Pedram's worst-case mapper and report the percentage improvement in average performance. It is very important to note that this comparison is not intended to yield an apple-to-apple measurement between comparable technology mappers but rather is intended to establish an estimate of how much our mapper can benefit from optimizing for the average case.

Table 3.6 compares the average-case performance of the two technology mappers. The results demonstrate that our average-case mapper can yield significant average-case performance improvements over the traditional worst-case mapper. The source of improvements can be decomposed in two major factors. First, we find and optimize only the true critical path delay for each pattern. Second, we find the probability for each pattern and prioritize the mapping for patterns with higher probabilities. Since the circuits MERGE, BUFCTRL1, Q42 and BINARY-COUNTER have uniform pattern frequencies, the obtained improvements can be attributed to the first factor. The other circuits have a more skewed distribution of pattern probabilities and thus the combination of both factors lead to significant improvement.

Table 3.7 shows that for 12 out of the 15 circuits tested our area is smaller than the area obtained using the worst-case mapper. This area reduction may be attributed to the fact that our algorithm uses large gates only on highly-frequent critical paths.

Note that for the worst-case mapped circuits, we optimized for delay (with no area constraints) and always picked points which met the required time propagated down from the primary output. Interestingly, the area and delay of the worst-case mapped circuits can sometimes be improved by choosing points that do not meet the required time but provide a better area-delay tradeoff. This same optimization can also be applied to the average-case mapped circuits, with similar gains to be expected. Because this optimization does not significantly reduce the overall average-case performance gains achieved by our circuits, the results are not shown.

3.4.4 Post-layout results

To further validate our results we integrated our techniques with the Mentor Graphics physical design tools and evaluated the automatically-generated layout using $0.8\mu m$ Government CMOSN standard cells. Specifically, we extracted the wire-capacitances from the layout and used them to analyze the impact on wire-delays. The results demonstrate that the wire capacitance changed the average performance by very little. Consequently, our post-layout improvements are similar in magnitude to our pre-layout values. The delay contribution of wire delay is small because the Mentor Graphic's router does a good job of keeping wires short for these relatively small circuits and because the standard cells in

Perf. of Average-Case and Worst-Case Optimized Circuits						
Circuits	AveLat Opt.			AveCyc Opt.		
	ACR	WC	Imp.	ACR	WC	Imp.
merge	1.82	1.97	7%	1.76	1.97	10%
bufctrl1	2.14	2.34	8%	2.54	2.60	2%
p3	1.01	1.09	7%	1.28	1.40	9%
q42	2.60	2.61	0%	2.77	2.93	5%
scsi-init-send	2.29	2.72	16%	2.29	2.72	16%
dramc	2.08	2.40	13%	2.16	2.40	10%
binary-counter	1.54	1.81	15%	2.95	3.63	19%
pe-send-ifc	2.74	3.56	23%	3.14	3.93	20%
trcv	3.57	4.33	18%	3.55	4.40	19%
ircv	3.17	3.89	18%	3.18	4.36	27%
tsend	3.92	4.90	20%	3.91	5.05	22%
p1	3.62	4.11	12%	4.20	4.87	14%
isend	4.55	4.96	8%	4.62	5.15	10%
cache-ctrl	5.34	6.02	11%	5.44	6.03	10%
scsi	4.21	5.61	25%	4.39	5.66	22%

Table 3.6: Average-case performance comparison between average-case and worst-case mappers. ACR corresponds to circuits optimized for the average-case with rotation. WC corresponds to circuits optimized for the worst-case. Delays are measured in nanoseconds.

Area of Average-Case and Worst-Case Optimized Circuits						
Circuits	AveLat Opt.			AveCyc Opt.		
	ACR	WC	Imp.	ACR	WC	Imp.
merge	11	15	29%	12	15	21%
bufctrl1	14	13	-3%	15	13	-10%
p3	15	20	23%	19	20	5%
q42	24	23	-2%	21	23	8%
scsi-init-send	29	32	10%	29	32	9%
dramc	64	68	7%	64	68	5%
binary-counter	76	78	3%	73	78	6%
pe-send-ifc	79	87	9%	80	87	8%
trcv	153	173	12%	152	173	12%
ircv	158	175	10%	162	175	7%
tsend	283	295	4%	292	295	1%
p1	333	350	5%	320	350	8%
isend	431	459	6%	432	459	6%
cache-ctrl	705	645	-9%	691	645	-8%
scsi	547	559	2%	554	559	1%

Table 3.7: Area(x10^3) comparison between average-case and worst-case mapped circuits. ACR corresponds to circuits optimized for the average-case with rotation. WC corresponds to circuits optimized for the worst-case.

the CMOSN library are relatively large which makes their intrinsic gate delay dominate the associated wire delay.

Experimental Results for Post-Layout Circuits						
Average Latency						
	Ave-Case		Worst-Case		Imp.	
Circuits	NCP	WCP	NCP	WCP	NCP	WCP
merge	0.71	0.73	0.76	0.78	6%	6%
bufctrl1	0.92	0.95	1.20	1.24	23%	23%
p3	0.47	0.49	0.50	0.52	5%	5%
q42	1.27	1.31	1.31	1.35	3%	3%
scsi-init-send	1.10	1.13	1.29	1.34	15%	15%
dramc	0.86	0.89	0.92	0.95	6%	6%
binary-counter	0.64	0.67	0.74	0.77	14%	14%
pe-send-ifc	1.34	1.39	1.58	1.65	16%	16%
trcv	1.52	1.58	1.67	1.73	9%	9%
ircv	1.48	1.54	1.98	2.06	25%	25%
tsend	1.70	1.77	2.00	2.09	15%	15%
p1	1.56	1.63	1.75	1.83	11%	11%
isend	1.70	1.78	1.89	1.97	10%	10%
cache-ctrl	1.70	1.78	1.93	2.03	12%	12%
scsi	1.33	1.39	1.71	1.79	22%	22%

Table 3.8: Experimental results comparing the average latency of average-case and worst-case mapped circuits including back-annotated wire-capacitance from post-layout circuits. WCP (NCP) corresponds to circuits with wiring capacitance (not) included. Delays are measured in nano-seconds.

Experimental Results for Post-Layout Circuits						
Average Cycle Time						
	Ave-Case		Worst-Case		Imp.	
Circuits	NCP	WCP	NCP	WCP	NCP	WCP
merge	0.71	0.73	0.76	0.78	6%	6%
bufctrl1	1.15	1.18	1.22	1.26	6%	6%
p3	0.59	0.61	0.67	0.69	12%	12%
q42	1.31	1.35	1.32	1.37	1%	1%
scsi-init-send	1.22	1.26	1.29	1.34	5%	6%
dramc	0.86	0.89	0.92	0.95	6%	6%
binary-counter	1.20	1.25	1.46	1.51	18%	18%
pe-send-ifc	1.49	1.55	1.76	1.83	16%	16%
trcv	1.59	1.66	1.67	1.74	5%	5%
ircv	1.38	1.44	1.99	2.06	31%	30%
tsend	1.76	1.84	2.07	2.16	15%	15%
p1	1.73	1.80	1.97	2.05	12%	12%
isend	1.81	1.90	1.93	2.01	6%	5%
cache-ctrl	1.72	1.80	1.94	2.04	11%	12%
scsi	1.47	1.53	2.07	2.16	29%	29%

Table 3.9: Experimental results comparing average cycle time of average-case and worst-case mapped circuits including back-annotated wire-capacitance from post-layout circuits. WCP (NCP) corresponds to circuits with wiring capacitance (not) included. Delays are measured in nano-seconds.

Chapter 4

Average-Case Mapping on One-Hot Domino Logic

We now describe how we can extend our technology mapping techniques to accommodate one-hot domino logic. In particular, we show how we perform technology mapping in the presence of incompletely-specified input patterns and the domino constraint.

4.1 Incompletely-specified patterns

An incompletely-specified pattern is a function from primary input variables to the set $\{0, 1, X\}$. The problem is that the delay of the circuit for an incompletely-specified pattern cannot be precisely defined because the exact set of gates that will evaluate is unknown in the presence of a primary input with the value "X". Moreover, since the exact set of evaluating gates is unknown, it is unclear which paths the technology mapper should optimize.

To address these problems, we interpret an incompletely-specified pattern as a collection of minterms over the input variables, where each minterm corresponds to a *compatible completely-specified pattern*. Formally, a completely-specified pattern is a function from primary inputs variables to the set $\{0, 1\}$. A completely-specified pattern i is compatible with an incompletely-specified pattern c if the assignments agree on all input variables not assigned to "X" in c.

It is clear that the circuit *pattern delay* for a compatible completely-specified pattern is well defined and can be established through simulation. The pattern delay, *p-delay*(p), of the one-hot domino circuit for pattern p is the pattern arrival time at the primary output which evaluates for pattern p. The pattern arrival time at each gate of the one-hot domino circuit can be computed recursively using Equation 3.1 and 3.2. Note that, since all fanin gates of a gate in domino logic precharge and evaluate to the same logic level (1 or 0), no opposite transitions will occur for the fanins of a gate and Equation 3.1 and 3.2 are the only two possible cases for computing pattern arrival times in domino logic. The objective of our mapping for one-hot domino circuits is to minimize *average-case delay* which is weighted sum of all pattern delays for all incompletely-specified input patterns, each of which is weighted by its corresponding probability.

Recall that we hope that we can define a new delay function which returns a delay value or bound for all compatible patterns of an incompletely-specified pattern when computing pattern delays for the objective. In principle, we are able to return both upper bound and lower bound of a pattern delay for an incompletely-specified pattern c such that a delay *range* for c is found. The *minimum* (*maximum*) of the range is the smallest (largest) pattern delay for all compatible patterns of c. Note that the number of compatible patterns can be exponential in the number of circuit variables. Thus, exhaustively simulating all compatible patterns is computationally very expensive.

Fortunately, the special nature of domino logic can be used to simplify this analysis. Specifically, this section proves that two easily identifiable compatible patterns, referred to as *representative* patterns, yield the lower and upper bounds of the pattern delay for an incompletely-specified pattern. The section then describes how we can use these representative patterns in technology mapping.

4.1.1 Pattern delay bound: intuition

The intuition behind our theory may be described with an analogy to the game called dominos (which is the origin of the name "domino logic"). In this game, rectangular tiles are often arranged in a linear fashion (or sometimes in more complex networks) such that the first tile falling causes a chain reaction of falling tiles. The delay of the chain reaction is the time in between the first and last tile falling. Notice that more than one tile can fall simultaneously to start the chain reaction and that some tiles may remain standing after the chain reaction completes.

Consider further the case where the set of tiles that start the reaction is not fully specified. In particular, consider the case where certain combinations of tiles can be chosen to start the chain reaction but the choice of which combination is unknown. In this case, the chain reaction delay cannot be determined. However, a lower bound on the chain reaction delay can be obtained by tipping over any tile which is tipped over in any combination.

Similarly, an upper bound on the chain reaction delay can be obtained by tipping over only those tiles which are tipped over in all combinations.

The analogy is that a domino gate is like a tile. We say a dynamic (static) gate *evaluates* if its output falls (rises). Gates that evaluate are like tiles that fall; they cannot return to their original value until the precharge phase. When one gate evaluates it can cause other gates to evaluate in what is like a chain reaction. Moreover, the evaluation delay is analogous to the delay of the chain reaction. Finally, an incompletely-specified input pattern is analogous to the situation where the set of tiles that starts the chain reaction is not fully specified.

Thus, to find a lower bound of the delay for an incompletely-specified pattern, we force any PI gate that evaluates under any compatible pattern to evaluate. Similarly, to find the upper bound of the delay we force only those PI gates that evaluates under all compatible patterns to evaluate.

In our application, the PI gates are restricted to be dynamic (see Section 2.3.1). Thus, to find the lower bound we set all unknown inputs to one. Similarly, to find the upper bound we set all unknown inputs to zero.

More formally, we define two *representative* patterns for an incompletely-specified pattern c. The *lower pattern* c_l is obtained by switching all $X's$ in c to 1 and yields a lower bound of c's pattern delay. Similarly, the *upper pattern* c_u is obtained by switching all $X's$ in c to 0 and yields an upper bound of c's pattern delay.

It is important to note that the bound is loose in the presence of dual-rail inputs a^T and a^F since in reality both a^T and a^F cannot be set to the same value.

4.1.2 Pattern delay bound: theory

This section formalizes our intuition.

In order to compute the evaluation delay for each pattern, we first re-define $N(i)$ in Chapter 3 as the set of all gates which evaluate when pattern i applied. Each gate's pattern arrival time can be still computed recursively using Equation 3.1 and 3.2. Note that since the circuit is one-hot encoded, any input pattern can make only one PO gate (any gate that drives a primary output) evaluate. Let $po(i)$ denote a function which returns the evaluating PO when pattern i is applied. The pattern delay of the circuit for pattern i, denoted *p-delay*(i), is equal to $pat(i, po(i))$.

In addition, let F_i denote the set of all PIs whose value is 1 when pattern i is applied. Moreover, let $FI(g)$ denote the set of all inputs of gate g and let $E(i,k)$ denote the set of all evaluating gates in level k of the circuit when pattern i is applied.

The following two lemmas prove our intuition that the representative patterns c_l (c_u) yields the lower (upper) bound of the delay for an incompletely-specified pattern c. Informally speaking, the first lemma proves that the more PIs set to one the more gates will evaluate and the second lemma proves that the more gates that evaluate the smaller the resulting pattern delay. Their proofs are given in the appendix.

Lemma 4.1.1 *If all PI gates are dynamic and $F_i \subseteq F_j$, then $E(i,l) \subseteq E(j,l)$ for every level l.*

Lemma 4.1.2 *If all PI gates are dynamic and $F_i \subseteq F_j$, then, for every level l and all $g \in E(i,l)$, we have that $pat(j,g) \leq pat(i,g)$.*

The following corollary follows directly from the application of Lemma 4.1.2 on the primary outputs from which it is easy to conclude our argument.

Corollary 4.1.1 *If all PI gates are dynamic, $F_i \subseteq F_j$ then p-delay$(j) \leq$ p-delay(i).*

Theorem 1 *Let c_l and c_u be the lower and upper pattern of an incompletely-specified input pattern c, respectively. Assuming all PI gates are dynamic, then for all c, p-delay(c_l) (p-delay(c_u)) is a lower (upper) bound of all pattern delays for all completely-specified patterns that are compatible with c.*

Proof: Consider a completely-specified pattern i that is compatible with c. Since pattern c_l (c_u) is generated by switching all $X's$ in pattern c to 1 (0), $F_{c_u} \subseteq F_i \subseteq F_{c_l}$. Therefore, according to Corollary 4.1.1, *p-delay*$(c_l) \leq$ *p-delay*$(i) \leq$ *p-delay*(c_u). □

4.1.3 Optimizing for representative patterns

As mentioned earlier, the technology mapping algorithms presented in Chapter 3 cannot handle input combinations described using incompletely-specified patterns. One means of working with incompletely-specified patterns is to optimize with respect to *all* compatible patterns. However, this has two problems. First, it is unknown how the probability of an incompletely-specified pattern is distributed over all of its compatible patterns. Thus, only approximate measures of overall pattern delay could be computed. Second, since the number of compatible patterns could be quite large, analyzing all compatible patterns independently can be computationally intractable.

In this chapter, we propose to optimize the circuit for one representative pattern for each incompletely-specified pattern. The choice of representative patterns is very important and different input representative patterns can lead to very different results.

In this chapter, we tested two sets of representative patterns to optimize for. For a set of incompletely-specified input patterns C, we define $L = \{c_l|$ for all $c \in C\}$ as the *lower set* of C, and $U = \{c_u|$ for all $c \in C\}$ as the *upper set* of C. We run the optimization procedure twice, once optimizing the benchmark for the lower set and once optimizing the benchmark for the upper set. Since the average-case delay is the weighted sum of all pattern delays for all incompletely-specified patterns, we can easily conclude that the average-case delay for the lower (upper) set is the lower (upper) bound of the average-case delay for the original incompletely-specified patterns. Therefore, for each of the two optimization results, we use the upper and lower sets again to obtain a range of average-case delay. Then, we let the user select the better result.

4.2 Handling the domino constraint

Recall that the input to the covering is a NAND-decomposed DAG, referred to as a *subject graph*. Our goal is to cover the subject graph with a set of library gates which are all inverting and either static or dynamic. Let $\langle N, E\rangle$ be a subject graph where N is a set of nodes and, E is a set of edges ($E \subseteq N \times N$).

Recall also, that one stage of domino logic can implement only monotonic logic. This limitation is manifested in technology mapping by the fact that not all decomposed networks can be mapped using domino logic. Consider the decomposed network in which there are two re-convergent fanout paths from u to v, where u is a gate driven by a primary input. Let the first be $u, n_1, n_2, \ldots, n_l, v$ and let the second be $u, n'_1, n'_2, \ldots, n'_{l'}, v$. If l and l' are both odd (both even) then the domino constraint demands that v is implemented with a dynamic (static) gate. If l is even and l' is odd (or vice-versa) then no mapping exists. Fortunately, this situation can be resolved by duplicating portions of the NAND-decomposed

Instruction prefix	Address-size prefix	Operand-size prefix	Segment override
0 or 1 Bytes	0 or 1 Bytes	0 or 1 Bytes	0 or 1 Bytes

Opcode	ModR/M	SIB	Displacement	Immediate
1 or 2 Bytes	0 or 1 Bytes	0 or 1 Bytes	0,1,2 or 4 Bytes	0,1,2 or 4 Bytes

Figure 4.1: The Pentium® instruction format.

network and introducing dual-rail inputs [80]. The result is an altered NAND-decomposed graph which is domino-feasible, as defined below.

Definition 4.2.1 (Domino-feasible DAG) *A domino-feasible DAG is a triple $\langle N, E, \lambda \rangle$, where N is a set of nodes and, E is a set of edges ($E \subseteq N \times N$) and λ is a labeling function $NI \rightarrow \{$Dynamic, Static$\}$ that satisfies $\lambda(u) = Dynamic$ for all $u \in I$ and $\lambda(u) \neq \lambda(v)$ if $(u, v) \in E$ where $u, v \in NI$.*

Then, to extend our technology mapping technique in to domino circuits, we simply restrict the matching of static (dynamic) nodes to only static (dynamic) gates. The remaining parts of the algorithm is the same as the techniques which are described in Chapter 3

4.3 A case study

We now describe the key combinational block of an asynchronous instruction length decoder (AILD) of Intel RAPPID project. The overall architecture of RAPPID and the associated control circuits are outside of the scope of this thesis and is reported in [62].

4.3.1 Instruction format

Figure 4.1 shows the general instruction format for the Pentium® processor [1]. Instructions consist of 4 optional instruction prefixes, opcode bytes, an optional address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, and optional displacement and immediate fields.

Each prefix is one byte long. Only the operand-size prefix and the address-size prefix affect the instruction length. Because these are very rare, we choose to trap and handle them using slower exception logic which will not be discussed here. The opcode represents the operation of the instruction. It identifies the size of the operation, the displacement, and the immediate. It is either one byte long or two bytes long where the first byte is always 0F. The ModR/M byte identifies a special addressing form for instructions that refer to an operand in memory. The ModR/M byte always follows the opcode. Some ModR/M bytes are followed by the SIB byte, a second addressing byte. ModR/M and SIB also determine the existence and size of the displacement and immediate. The displacement follows the opcode, or ModR/M, or SIB (which ever is last). The immediate, if present, is always the last field of an instruction. Both the displacement and immediate fields can be one, two, or four bytes long. The maximum valid instruction length is 15 bytes. Figure 4.2 shows the ModR/M and SIB byte format. The details of each field can be found in [1].

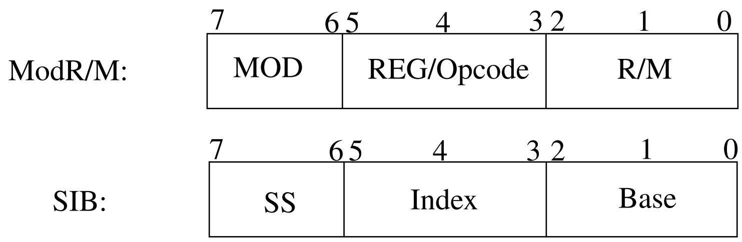

Figure 4.2: The ModR/M and SIB fields.

4.3.2 Instruction length frequencies

The motivation of the asynchronous design stems from an analysis of several benchmark programs in which instruction lengths are monitored. This analysis led to the frequency histogram presented in Figure 4.3. This chart clearly shows that instructions of lengths two and three are very frequent, whereas others are much less frequent. Instructions of length greater than seven are extremely rare. This motivates our design to be optimized for instructions of length 7 or less. Longer instructions are handled separately using slower logic that is not discussed here.

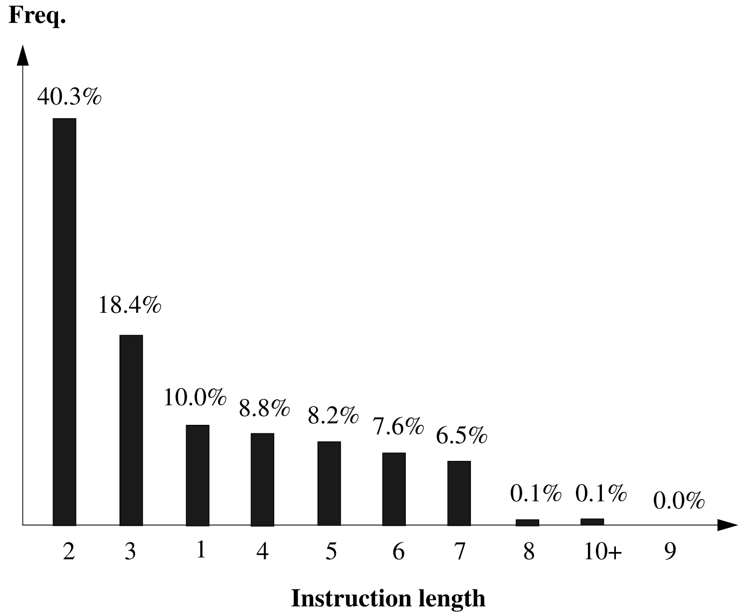

Figure 4.3: The frequency of instruction lengths.

4.3.3 One-hot domino logic blocks

One-hot domino logic forms the combinational block that inputs an instruction and yields the one-hot encoded instruction length for the instructions with lengths less than 7. Specifically, as shown in Figure 4.4, this block is decomposed into 6 one-hot domino logic blocks: Opcode1, Opcode2, Mem1, Mem2, and two length merging blocks, Merge1 and Merge2. The Opcode1 and Opcode2 blocks compute the length contributed by the first and second opcode byte, respectively. The Mem1 and Mem2 blocks compute the length contributed by the ModR/M byte for one-byte and two-byte opcodes, respectively. The two merging blocks add these contributions to form the final length outputs.

The Opcode1 block generates the following 11 one-hot encoded outputs: Op1O1NoM, Op1Oc2M1, Op1O2NoM, Op1Oc3M1, Op1O3NoM, Op1Oc4M1, Op1O4NoM, Op1O5NoM, Op1Oc6M1, Op1O7NoM, and is0F. Op1Oc2M1, for example, denotes that the first byte of the instruction is the only opcode byte and it contributes two bytes for the total length and the ModR/M byte is present. Op1O2NoM denotes the same information as Op1Oc2M1 except that no ModR/M byte is present. The other outputs have similar interpretations. Note that Op1O6NoM, for example, is not possible. The is0F output is asserted when the opcode consists of two bytes (in which case the first byte must be 0F).

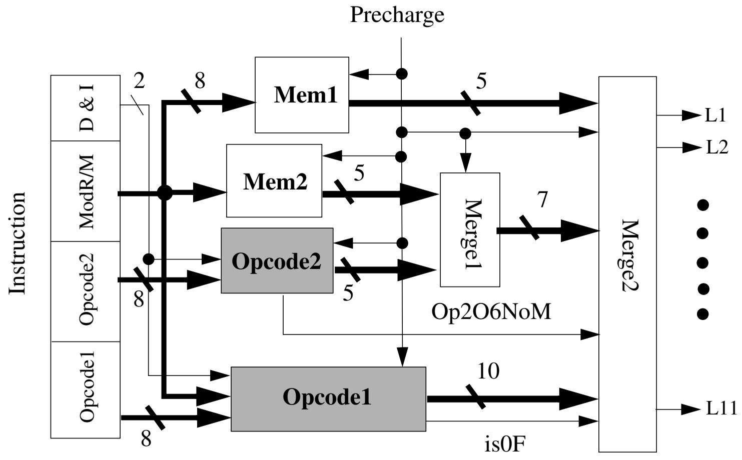

Figure 4.4: The block diagram of the asynchronous instruction length decoder.

The Opcode2 block generates 6 one-hot encoded outputs defined similarly. Op2Oc3M2, for example, denotes that the second opcode byte contributes three bytes (including the first opcode byte 0F) and the ModR/M byte is present.

The Mem1 (Mem2) block checks the ModR/M byte for the one-byte (two-byte) opcode to generate 5 one-hot encoded outputs: M10, M11, M12, M14, and M15 (M20, M21, M22, M24, and M25). These represent that the ModR/M byte contributes 0, 1, 2, 4, and 5 bytes for the total length, respectively.

The Merge1 block combines the Opcode2's outputs (except Op2O6NoM) and the Mem2's outputs to obtain the length for the instructions having a two-byte opcode (see Table 4.1). The Merge2 block then combines the outputs of the Opcode1, Mem1, and Merge1 (along with the Op2O6NoM from the Opcode2) to obtain the final one-hot length outputs, as defined in Table 4.2. This configuration means that the instructions having a two-byte opcode will have longer length computation time than the instructions having a one-byte opcode except the one represented by the Op2O6NoM. This improves the average-case delay of the length computation because most one-byte-opcode instructions are more frequent than the two-byte-opcode instructions. The Op2O6NoM is chosen to be fed directly to the Merge2 since it is also frequent and it need not be ANDed with any Mem2's output.

L	Out	Equation
3	L3_0F	Op2Oc3M2*M20 + Op2O3NoM
4	L4_0F	Op2Oc3M2*M21 + Op2Oc4M2*M20 + Op2O4NoM
5	L5_0F	Op2Oc3M2*M22 + Op2Oc4M2*M21
6	L6_0F	Op2Oc4M2*M22
7	L7_0F	Op2Oc3M2*M24
8	L8_0F	Op2Oc3M2*M25 + Op2Oc4M2*M24
9	L9_0F	Op2Oc4M2*M25

Table 4.1: The length equations implemented in the Merge1 block.

L	Out	Equation
1	L1	Op1O1NoM
2	L2	Op1Oc2M1*M10 + Op1O2NoM
3	L3	Op1Oc2M1*M11 + Op1Oc3M1*M10 + Op1O3NoM + is0F*L3_0F
4	L4	Op1Oc2M1*M12 + Op1Oc3M1*M11 + Op1Oc4M1*M10 + Op1O4NoM + is0F*L4_0F
5	L5	Op1Oc3M1*M12 + Op1Oc4M1*M11 + Op1O5NoM + is0F*L5_0F
6	L6	Op1Oc2M1*M14 + Op1Oc4M1*M12 + Op1Oc6M1*M10 + is0F*Op2O6NoM + is0F*L6_0F
7	L7	Op1Oc2M1*M15 + Op1Oc3M1*M14 + Op1Oc6M1*M11 + Op1O7NoM + is0F*L7_0F
8	L8	Op1Oc3M1*M15 + Op1Oc4M1*M14 + Op1Oc6M1*M12 + is0F*L8_0F
9	L9	Op1Oc4M1*M15 + is0F*L9_0F
10	L10	Op1Oc6M1*M14
11	L11	Op1Oc6M1*M15

Table 4.2: The length equations implemented in the Merge2 block.

4.3.4 Product term frequencies

For each combinational logic block, a two-level minimizer is used to obtain an optimized set of product terms. Then, architectural simulations is used to obtain frequency statistics of each product term. We then associate with each product term an incompletely-specified pattern and use the normalized product-term frequencies as an estimate of the frequency of the incompletely-specified pattern. The resulting frequency distributions of the incompletely-specified patterns for the Op1O1NoM and Op1Oc2M1 outputs are given in Figure 4.5.

The distributions of all patterns for all outputs of both the Opcode1 and Opcode2 blocks, along with the output's optimized NAND-decomposed network, are then input to our technology mapping program.

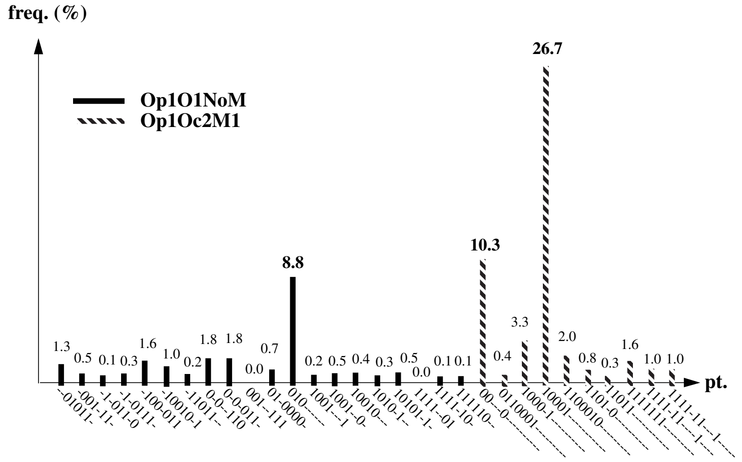

Figure 4.5: The frequency distribution of product terms of Op1O1NoM and Op1Oc2M1. The first opcode byte and the ModR/M byte are inputs of the product terms. For Op1O1NoM, the inputs from the ModR/M byte are don't-cares (not shown for simplicity).

4.3.5 Experimental results

This section reports the technology mapping results for both the Opcode1 and Opcode2 blocks which are the shaded blocks in Figure 4.4. A summary of the complexity of each output logic is given in Table 4.3. Notice that the fourth column reports the number of incompletely-specified input patterns which cause the output to evaluate to 1. The fifth

Description of each combinational logic output					
Circuit	# PIs	# POs	# Patts.	# Nodes	Freq.
Op1O1NoM	16	1	20	574	0.202
Op1Oc2M1	16	1	10	321	0.473
Op1O2NoM	16	1	9	322	0.114
Op1Oc3M1	18	1	6	280	0.065
Op1O3NoM	18	1	7	307	0.018
Op1Oc4M1	18	1	4	231	0.004
Op1O4NoM	8	1	1	56	0.001
Op1O5NoM	19	1	8	361	0.056
Op1Oc6M1	18	1	4	227	0.015
Op1O7NoM	12	1	2	112	0.000
Op2O2NoM	16	1	16	427	0.001
Op2Oc3M2	16	1	15	379	0.025
Op2O3NoM	6	1	1	42	0.000
Op2Oc4M2	11	1	2	95	0.001
Op2O4NoM	11	1	2	74	0.003
Op2O6NoM	5	1	1	33	0.022

Table 4.3: Summary of complexity of each combinational logic output.

column reports the number of nodes in the NAND-decomposed DAG. The sixth column reports the relative frequency of each output evaluating to a 1.

Note that all mappings are performed using the *lib2* gate library (that is available in the tool SIS [66]) which is modified in two ways. First, we remove all non-inverting gates because such gates cannot be used in domino logic. Second, for each inverting static gate, we add a corresponding dynamic gate with the same area and delay characteristics. This made it possible for us to compare our results with those obtained using worst-case mapping techniques that do not ensure the domino constraint [11]. All experiments were performed on a 120-MHz Pentium® Processor with manageable CPU times.

Table 4.4 reports the average-case delays obtained by optimizing the logic for both the lower set (the 2nd and 3rd columns) and the upper set (the 4th and 5th columns). Not surprisingly, the results indicate that when we optimize for the lower pattern set, the lower bound is typically smaller than when we optimize for the upper pattern set. Similarly, optimizing for the upper pattern set leads to smaller upper bounds. When comparing circuits, we always try to be conservative and thus report the upper bound of our circuits. Consequently, it appears that optimizing the upper bound of our circuits generally leads to more favorable conservative comparisons.

Interestingly, the ranges in average-case delay obtained by optimizing for the upper pattern set are always smaller than those obtained by optimizing for the lower pattern set. This may be because the critical path for an upper pattern with high frequency is typically very short because it has been highly optimized. Consequently, when the corresponding lower pattern is applied, the path is still critical. On the other hand, when we optimize for the lower pattern set, we may optimize for a critical path that is different from the one that is critical for the upper pattern, thereby yielding a large range.

Table 4.4 also presents data derived from circuits obtained using the worst-case mapping techniques described in [11] (columns 7-10). Using this data we can make two major comparisons.

First, we compare the average-case delay of our best circuits (optimized for the upper pattern set) with the average-case delay of circuits obtained with worst-case mapping techniques. This is of interest because it establishes the potential benefit of explicitly optimizing average-case delay during technology mapping. To be conservative, we compare the upper bound of our mapped circuits with the lower bound of the circuits derived using worst-case mapping techniques. The results demonstrate that our circuits are at least 31% faster on average than that of worst-case mapped circuits.

Second, we can compare the average-case delay of our circuits with the worst-case delay of the comparable synchronous circuit. This comparison can give us an estimate of the potential benefit of asynchronous circuits. It is important to note, however, that this estimate assumes that the synchronous circuit adopts the same decomposition of blocks that is described here. Specifically, we cannot account for the possibility that a different decomposition might be better suited for optimizing for worst-case delay. With this caveat stated, the results indicate that our circuits are on average at least 54% faster than the comparable synchronous circuits. This demonstrates the potential advantage of asynchronous one-hot domino circuits over both synchronous implementations and conventional bundled-data implementations

Average-case Mapping vs. Worst-case Mapping											
Circuit	Average-case (AC)					Worst-case (WC)				Improve	
	$ACD_u^{a,l}$	$ACD_l^{a,l}$	$ACD_u^{a,u}$	$ACD_l^{a,u}$	Area^{a,u}	ACD_u^w	ACD_l^w	WCD	Area^w	AA	AW
Op1O1NoM	3.672	1.570	2.965	2.229	114144	3.802	3.279	5.130	97904	10%	42%
Op1Oc2M1	2.404	2.131	2.186	2.018	55680	3.510	3.459	4.070	55216	37%	46%
Op1O2NoM	1.775	1.698	1.811	1.800	54752	4.012	4.002	4.440	51040	55%	59%
Op1Oc3M1	2.201	2.047	2.170	2.070	43616	3.206	3.151	4.010	46400	31%	46%
Op1O3NoM	2.821	2.821	2.821	2.821	50576	3.542	3.542	4.200	49648	20%	33%
Op1Oc4M1	2.761	2.761	2.761	2.761	33872	3.104	3.104	3.370	39440	11%	18%
Op1O4NoM	1.587	1.587	1.587	1.587	6032	1.587	1.587	1.590	6032	0%	0%
Op1O5NoM	2.800	2.800	2.800	2.800	61248	3.516	3.516	4.200	61248	20%	33%
Op1Oc6M1	2.647	2.647	2.647	2.647	37584	3.181	3.181	3.370	39440	17%	21%
Op1O7NoM	2.400	2.400	2.400	2.400	14384	2.400	2.400	2.750	14384	0%	13%
Ave.(Op1)	2.620	2.017	2.364	2.114	471888	3.604	3.462	5.130	460752	32%	54%
Op2O2NoM	4.391	1.810	2.150	2.043	77024	4.270	4.181	4.790	70528	49%	55%
Op2Oc3M2	3.206	1.507	2.567	2.224	74704	3.624	3.137	4.680	65424	18%	45%
Op2O3NoM	1.400	1.400	1.400	1.400	5104	1.400	1.400	1.440	5104	0%	3%
Op2Oc4M2	1.675	1.675	1.675	1.675	12064	2.403	2.403	2.410	12064	30%	30%
Op2O4NoM	1.537	1.537	1.537	1.537	10208	1.627	1.627	2.070	9280	6%	26%
Op2O6NoM	1.335	1.335	1.335	1.335	4640	1.335	1.335	1.330	4640	0%	0%
Ave.(Op2)	2.311	1.445	1.962	1.794	183744	2.530	2.293	4.790	167040	14%	59%
Ave.(Op1+2)	2.604	1.987	2.343	2.098	655632	3.548	3.401	5.130	627792	31%	54%

Table 4.4: Delay and area of average-case mapping vs. delay and area of worst-case mapping. ACD denotes the average-case delay while WCD denotes the worst-case delay. Subscripts and superscripts on ACD and Area denote the type of optimization performed and the bound of the average-case delay reported. Specifically, the superscript a denotes the use of our average-case mapper while w denotes the use of the worst-case mapper. The superscripts u and l denote the optimization is performed for the upper set and the lower set, respectively. In contrast, the subscripts u and l denote the numbers reported are the upper and lower bound of the average-case delay, respectively. For the percentage improvements, the numbers in column AA are computed using (1-$ACD_u^{a,u}$/ACD_l^w)*100%, and the numbers in column AW are computed using (1-$ACD_u^{a,u}$/WCD)*100%.

Appendix: Proof of lemmas

Lemma 4.4.1 *If all PI gates are dynamic and $F_i \subseteq F_j$, then $E(i,l) \subseteq E(j,l)$ for every level l.*

Proof: (By induction)
Base: Let $l = 1$. $F_i \subseteq F_j$. Since all PI gates are dynamic, $E(i,1) \subseteq E(j,1)$.
Inductive hypothesis: For $l = k$, $E(i,k) \subseteq E(j,k)$.
Inductive step: Let $l = k+1$. Let $g_{k+1} \in E(i,k+1)$. First consider the case where g_{k+1} has a controlling fanin $f_k \in FI(g_{k+1})$ which is driven by a gate g_k that evaluates when i is applied. Since $E(i,k) \subseteq E(j,k)$, g_k must evaluate in pattern j. Since the controlling nature of an input is pattern-independent (because an evaluating gate always drives its output to a value that is independent of the pattern applied), f_k must also be a controlling input of g_{k+1} when j is applied. Therefore, g_{k+1} must evaluate when j is applied, i.e., $g_{k+1} \in E(j,k+1)$. Thus, $E(i,k+1) \subseteq E(j,k+1)$.

Now consider the case where g_{k+1} evaluates and all $f_k \in FI(g_{k+1})$ are non-controlling fanins of g_{k+1} in pattern i. Since $E(i,k) \subseteq E(j,k)$, all g_k's must evaluate and all corresponding f_k's must be non-controlling fanins of g_{k+1} when j is applied. Thus, g_{k+1} must evaluate when j is applied, i.e., $g_{k+1} \in E(j,k+1)$. Therefore, $E(i,k+1) \subseteq E(j,k+1)$. $\square$

Lemma 4.4.2 *If all PI gates are dynamic and $F_i \subseteq F_j$, then, for every level l and all $g \in E(i,l)$, we have that $pat(j,g) \leq pat(i,g)$.*

Proof: (By induction)
Base: Let $l = 1$. Since $F_i \subseteq F_j$, according to Lemma 4.1.1, $E(i,1) \subseteq E(j,1)$. For $g \in E(i,1)$, two conditions that $FC(i,g) \subseteq FC(j,g)$ and $FNC(i,g) = FNC(j,g)$ must hold. Thus, $pat(g,j) \leq pat(g,i)$.
Inductive hypothesis: For $l = k$, and for all $g_k \in E(i,k)$, $pat(g_k,j) \leq pat(g_k,i)$.
Inductive step: Let $l = k+1$. Consider $g_{k+1} \in E(i,k+1)$.

Case 1: g_{k+1} has a controlling fanin. According to Equation 3.1, $pat(g_{k+1},i) = \min_{f_k \in FC(i,g_{k+1})} pat(g_k,i) +$ *p2p-delay*(g_{k+1},f_k,i), and $pat(g_{k+1},j) = \min_{f_k \in FC(j,g_{k+1})} pat(g_k,j) +$ *p2p-delay*(g_{k+1},f_k,j). According to Lemma 4.1.1, since g_k evaluates when i is applied, g_k must evaluate when j is applied. Since f_k is a controlling fanin of g_{k+1} in i, we know that it must be a controlling fanin of g_{k+1} in j. Thus, $FC(i, g_{k+1}) \subseteq FC(j, g_{k+1})$. By the inductive hypothesis we know that $pat(g_k, j) \leq pat(g_k, i)$. Moreover, we know that the pin-to-pin delay of an evaluating gate is pattern independent, i.e., *p2p-delay*(g_{k+1}, f_k, i) = *p2p-delay*(g_{k+1}, f_k, j). Therefore, we conclude that $pat(g_{k+1}, j) \leq pat(g_{k+1}, i)$.

Case 2: g_{k+1} has only non-controlling fanins. From Equation 3.2, $pat(g_{k+1}, i) = \max_{f_k \in FNC(i, g_{k+1})} pat(g_k, i) +$ *p2p-delay*(g_{k+1}, f_k, i), and $pat(g_{k+1}, j) = \max_{f_k \in FNC(j, g_{k+1})} pat(g_k, j) +$ *p2p-delay*(g_{k+1}, f_k, j). According to Lemma 4.1.1, all g_k's that evaluate in i must evaluate in j. Since all f_k's are non-controlling in i they must be non-controlling in j. Therefore, $FNC(i, g_{k+1}) = FNC(j, g_{k+1})$ must hold. Also, by the inductive hypothesis we know that $pat(g_k, j) \leq pat(g_k, i)$. Moreover, we know that *p2p-delay*$(g_{k+1}, f_k, i) =$ *p2p-delay*(g_{k+1}, f_k, j). Thus, we conclude that $pat(g_{k+1}, j) \leq pat(g_{k+1}, i)$. $\square$

Chapter 5

Conclusions and Future Work

We presented a technology mapping technique for optimizing the average performance of asynchronous burst-mode control circuits and one-hot domino circuits. We use stochastic techniques to determine the frequency of each state transition for burst-mode circuits and use architectural simulations to determine the frequency of each interested input patterns for one-hot domino circuits. We incorporate these statistical information of the occurrence of input patterns with known analysis, decomposition, and covering techniques to optimize the mapped circuits for average latency/cycle time of burst-mode circuits and for average-case delay of one-hot domino circuits. We test our burst-mode mapping technique on 3D burst-mode benchmark and apply our one-hot domino mapping technique to two combinational logic blocks of the asynchronous instruction length decoder in Intel RAPPID project. Both results demonstrate that our techniques can simultaneously lead to significantly higher average performance and significantly smaller area when compared to traditional techniques.

We believe that our burst-mode technology mapping algorithms are applicable to Nowick's UCLOCK method [55]. However, extensions to fundamental-mode circuits implemented with generalized C-elements [86] is not trivial and is an interesting area of future research. Using generalized C-elements often increases the performance of the circuits, but their optimization would most likely involve handling hazard-free sequential decomposition as well as transistor-level delay analysis. Another possible direction for future work is to extend this type of average-case optimization to other control circuit design styles (e.g., quasi-delay insensitive, speed-independent, timed, etc.) or other levels of the design hierarchy (e.g., architectural and logic synthesis, placement and routing, fanout optimization, etc.). For one-hot domino mapping, possible future work is to explore the mapping technique for other domino design styles, such as self-resetting domino [7] and clock-delayed domino [84, 56].

Lastly, it may be possible to apply our average-case methodology to the technology mapping of synchronous telescopic units [5]. These blocks of combinational logic have variable latency (they either take one or two clock cycles) and mapping them for the average case can increase the probability that the unit takes only one clock cycle, thereby increasing system performance.

Reference List

[1] Intel architecture software developer's manual, volume 2: Instruction set reference manual. http://developer.intel.com/design.

[2] P. A. Beerel and T.H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992.

[3] P.A. Beerel and T.H.-Y. Meng. Semi-modularity and testability of speed-independent circuits. *Integration, the VLSI journal*, 13(3):301–322, September 1992.

[4] M. Benes, A. Wolfe, and S.M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, Los Alamitos, CA, september 1997. IEEE Computer Society Press.

[5] L. Benini, E. Macii, M. Poncino, and G. De Micheli. Telescopic units: a new paradigm for performance optimization of vlsi designs. *IEEE Transactions on Computer-Aided Design*, 17(3):220–232, March 1998.

[6] J. Benkoski, E. Vanden Meersch, L. J. M. Claesen, and H. De Man. Timing verification using statically sensitizable paths. *IEEE Transactions on Computer-Aided Design*, 9:1073–1084, Sept. 1990.

[7] K. Bernstein, K. M. Carrig, C. M. Durham, P. R. Hansen, D. Hogenmiller, E. J. Nowak, and N. J. Rohrer. *High Speed CMOS Design Styles*. Kluwer Academic Publishers, 1998.

[8] N. B. Bhat and D. D. Hill. Routable technology mapping for LUT FPGA's. In *Proc. International Conf. Computer Design (ICCD)*, pages 95–98, 1992.

[9] S. M. Burns. General conditions for the decomposition of state holding elements. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 48–57. IEEE Computer Society Press, March 1996.

[10] S. Chakraborty, D. L. Dill, K. Y. Chang, and K. Y. Yun. Timing analysis for extended burst-mode circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[11] K. Chaudhary and M. Pedram. Computing the area versus delay trade-off curves in technology mapping. *IEEE Transactions on Computer-Aided Design*, pages 1480–1489, December 1995.

[12] C. S. Chen, Y. W. Tsay, T. T. Hwang, A. C. H. Wu, and Y. L. Lin. Combining technology mapping and placement for delay optimization. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 123–127, 1993.

[13] H. Chen and D. Du. Path sensitization in critical path problem. *IEEE Transactions on Computer-Aided Design*, 12(2):196–207, February 1993.

[14] T. A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[15] T.-A. Chu, C. K. C. Leung, and T. S. Wanuga. A design methodology for concurrent VLSI systems. In *Proc. International Conf. Computer Design (ICCD)*, pages 407–410. IEEE Computer Society Press, 1985.

[16] J. Cong and Y. Ding. An optimal technology mapping for delay optimization in lookup-table based FPGA's designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 48–53, 1992.

[17] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev. Decomposition and technology mapping of speed-independent circuits using boolean relations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 220–227, 1997.

[18] A. Davis, B. Coates, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 409–418. IEEE Computer Society Press, January 1993.

[19] A. Davis and S. M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[20] S. C. Chang et al. Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams. In *IEEE Transactions on Computer-Aided Design*, pages 1226–1236, 1996.

[21] R. J. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast technology mapping for lookup table-based FPGA's. In *Proc. ACM/IEEE Design Automation Conference*, pages 613–619, 1991.

[22] R. J. Francis, J. Rose, and Z. Vranesic. Technology mapping for lookup table-based FPGA's for performance. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 568–571, 1991.

[23] S. B. Furber. Computing without clocks: Micropipelining the ARM processor. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 211–262. Springer-Verlag, 1995.

[24] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *Proceedings IEEE Computer Conference (COMPCON)*, pages 476–485, March 1994.

[25] S. B. Furber, J. D. Garside, and D. A. Gilbert. AMULET3: A high-perfromance self-timed ARM microprocessor. In *Proc. International Conf. Computer Design (ICCD)*, 1998.

[26] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. AMULET2e: An asynchronous embedded controller. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[27] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[28] R. Hitchcock, G. Smith, and D. Cheng. Timing analysis of computer hardware. *IBM journal of Research and Development*, 26(1):100–105, January 1982.

[29] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.

[30] S. Iman and M. Pedram. POSE: Power optimzation and synthesis environment. In *Proc. ACM/IEEE Design Automation Conference*, pages 21–26, 1996.

[31] S. Iman, M. Pedram, and K. Chaudhary. Technology mapping using fuzzy logic. In *Proc. ACM/IEEE Design Automation Conference*, pages 333–338, 1994.

[32] J. Kessels and P. Marston. Designing asynchronous standby circuits for a low-power pager. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[33] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *24th Design Automation Conference*, pages 341–347. IEEE/ACM, 1987.

[34] A. Kondratyev, M. Kishinevsky, J. Cortadella, L. Lavagno, and A. Yakovlev. Technology mapping for speed-independent circuits: Decomposition and resynthesis. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 240–253, 1997.

[35] D. Kung. Path sensitization in hazard-free circuits, 1995. In collection of papers of the *ACM International Workshop on Timing Issues in the Specification of and Synthesis of Digital Systems*.

[36] D. S. Kung. Hazard-Non-Increasing Gate-Level Optimization Algorithms. In *IEEE ICCAD Digest of Technical Papers*, pages 631–634, 1992.

[37] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Synthesis of hazard-free asynchronous circuits with bounded wire delays. *IEEE Transactions on Computer-Aided Design*, 14(1):61–86, January 1995.

[38] J. Lou, A. Salek, and M. Pedram. An exact solution to simultaneous technology mapping and linear placement problem. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1997.

[39] F. Mailhot and G. De Micheli. Algorithms for technology mapping based on binary decision diagrams and on Boolean operators. *IEEE Transactions on Computer-Aided Design*, 12(6):599–620, 1993.

[40] A. J. Martin. A synthesis method for self-timed VLSI circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 224–229, Rye Brook, NY, 1987. IEEE Computer Society Press.

[41] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[42] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[43] P. McGeer and R. Brayton. Efficient algorithms for computing the longest viable path in a combinational circuit. In *Proc. ACM/IEEE Design Automation Conference*, pages 561–567, 1989.

[44] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[45] C. E. Molnar, T. P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[46] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[47] M. Murgai, Y. Nishizaki, N. Shenoy, and R. K. Brayton. Logic synthesis algorithms for programmable gate arrays. In *Proc. ACM/IEEE Design Automation Conference*, pages 620–625, 1990.

[48] M. Murgai, N. Shenoy, and R. K. Brayton. Performance directed synthesis for table look up for programmable gate arrays. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 572–575, 1991.

[49] C. J. Myers. *private communication*, September 1998. Chris J. Myers is an assistant professor at University of Utah.

[50] C. J. Myers, P. A. Beerel, and T. H.-Y. Meng. Technology-mapping of timed-circuits. In *2nd Working Conference on Asynchronous Design Methodologies*, May 1995.

[51] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[52] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[53] S. M. Nowick. Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings, Part E, Computers and Digital Techniques*, 143(5):301–307, September 1996.

[54] S. M. Nowick, K. Y. Yun, and P. A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[55] S.M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *Proceedings of the 1994 IEEE International Conferen ce on Computer Design: VLSI in Computers and Processors*, October 1994.

[56] K. J. Nowka and T. Galambos. Circuit design techniques for a gigahertz integer microprocessor. In *International Conference on Computer Design*, pages 11–16. IEEE, 1998.

[57] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, June 1994.

[58] M. Pedram and N. Bhat. Layout driven technology mapping. In *Proc. ACM/IEEE Design Automation Conference*, pages 99–102, 1991.

[59] S. Perremans, L. Claesen, and H. DeMan. Static timing analysis of dynamically sensitizable paths. In *Proc. ACM/IEEE Design Automation Conference*, pages 568–573, 1989.

[60] R. Puri, A. Bjorksten, and T. E. Rosser. Logic optimization by output phase assignment in dynamic logic synthesis. In *International Conference on Computer-Aided Design*, pages 2–8. IEEE, 1996.

[61] S. Ross. *Introduction to Probability Models*. Academic Press, 1985.

[62] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: An asynchronous instruction length decoder. IEEE Computer Society Press, April 1999. in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*.

[63] J. P. Roth. Diagnosis of automata failures: A calculus and a new method. *IBM Journal of Research and Development*, pages 278–281, October 1966.

[64] R. Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, U. C. Berkeley, April 1989. Memorandum UCB/ERL M89/49.

[65] M. Schlag, J. Kong, and P. K. Chan. Routability-driven technology mapping for lookup table-based FPGA's. In *Proc. International Conf. Computer Design (ICCD)*, pages 86–90, 1992.

[66] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[67] P. Siegel and G. De Micheli. Decomposition methods for library binding of speed-independent asynchronous designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 558–565, November 1994.

[68] P. Siegel, G. De Micheli, and D. Dill. Automatic technology mapping for generalized fundamental-mode asynchronous designs. In *Proc. ACM/IEEE Design Automation Conference*, pages 61–67, June 1993.

[69] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[70] T. Thorp, G. Yee, and C. Sechen. Domino logic synthesis using complex static gates. In *International Conference on Computer-Aided Design*, pages 242–247. IEEE, 1998.

[71] T. Thorp, G. Yee, and C. Sechen. Dynamic logic synthesis using alternating dynamic and static gates, 1998. In collection of papers of the *ACM International Workshop on Timing Issues in the Specification of and Synthesis of Digital Systems*.

[72] V. Tiwari, P.Ashar, and S. Malik. Technology mapping for low power. In *Proc. ACM/IEEE Design Automation Conference*, pages 74–79, 1993.

[73] N. Togawa, M. Sato, and T. Ohtsuki. Maple: A simultaneous technology mapping, placement, and global routing algorithm for field-programmable gate arrays. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 156–163, 1994.

[74] H. J. Touati, C. W. Moon, R. K. Brayton, and A. Wang. Performance-oriented technology mapping. In W. J. Dalley, editor, *6th MIT Conference on Advanced VLSI Conference*, pages 79–97, 1995.

[75] C.-Y. Tsui, M. Pedram, and A. M. Despain. Power efficient technology decomposition and mapping under an extended power consumption model. *IEEE Transactions on Computer-Aided Design*, 13(9):1110–1122, 1995.

[76] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[77] K. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[78] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[79] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439, December 1994.

[80] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, 2nd edition, 1993.

[81] T. E. Williams. Dynamic logic: Clocked and asynchronous, 1996. ISSCC Tutorial.

[82] T. E. Williams and M. A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[83] H. Yasng and D. F. Wong. Edge-map: Optimal performance driven technology mapping for iterative lut based FPGA designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 150–155, 1994.

[84] G. Yee and C. Sechen. Dynamic logic synthesis. In *IEEE Custom Integrated Circuits Conference*, pages 345–348. IEEE, 1997.

[85] K. Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems*. PhD thesis, Stanford University, August 1994.

[86] K. Y. Yun. Automatic synthesis of extended burst-mode circuits using generalized C-elements. In *Proc. European Design Automation Conference (EURO-DAC)*, September 1996.

[87] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, J. Arceo, and Dhiraj K. Pradhan. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Transactions on VLSI Systems*, 6(4):643–655, December 1998.

[88] K. Y. Yun and D. L. Dill. Automatic synthesis of extended burst-mode circuits: part I (specification and hazard-free implementations). *IEEE Transactions on Computer-Aided Design*, 18(2):101–117, Febuary 1999.

[89] K. Y. Yun and D. L. Dill. Automatic synthesis of extended burst-mode circuits: part II (automatic synthesis). *IEEE Transactions on Computer-Aided Design*, 18(2):118–132, Febuary 1999.

[90] K. Y. Yun, A. E. Dooply, J. Arceo, P. A. Beerel, and V. Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[91] M. Zhao and S. S. Sapatnekar. Technology mapping for domino logic. In *International Conference on Computer-Aided Design*, pages 248–251. IEEE, 1998.

[92] H. Zhou and D. F. Wong. An exact gate decomposition algorithm for low-power technology mapping. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 575–580, 1997.