PERFORMANCE ENHANCEMENT TECHNIQUES FOR
PHASED LOGIC CIRCUITS

Approved by:

Dr. Mitchell A. Thornton

Dr. V. S. S. Nair

Dr. David W. Matula

PERFORMANCE ENHANCEMENT TECHNIQUES FOR PHASED LOGIC CIRCUITS

A Thesis Presented to the Graduate Faculty of the

School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Science

with a

Major in Computer Engineering

by

Kenneth B. Fazel

(B.S., Mississippi State University, 2002)

May 15, 2004

Fazel, Kenneth B. B.S., Mississippi State University, 2002

Performance Enhancement Techniques for
Phased Logic Circuits

Advisor: Associate Professor Mitchell A. Thornton
Master of Science degree conferred May 15, 2004
Thesis completed March 15, 2004

In this thesis, a presentation of automated techniques for speed up mechanisms in the asynchronous design methodology known as Phased Logic is given. In previous work, these mechanims have been added to Phased Logic circuits by ad-hoc and manual means that require a designers intimate knowledge of the circuitry. One of the benefits of Phased Logic is that it allows conventional synchronous designers to implement self-timed designs without a major shift to other design strategies, thus such automatic techniques are needed. Various methods for inserting Early Evaluation and slack matching buffers are presented, along with experimental results. Most notably, automatic Early Evaluation insertion results in up to 40% decrease of delay in some microprocessor designs. The slack matching buffer insertion results indicate that throughput performance may increase by 20% to 40% for certain architectures. Moreover, a more formal treatment of Early Evaluation is presented. A tool for visualizing token flow through a PL netlist is also presented.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am indebted to Prof. M.A. Thornton for his patience and guidance. His generosity has enabled me to pursue great oppurtunities. I would also like to thank Prof. R.B. Reese for his support in my academic endeavors. I appreciate the efforts of my thesis committee members for ensuring the quality of my thesis, however, any mistakes that may have escaped observation are of course my own.

Of course, I am most appreciative to my family, Mom, Dad, and Robert, for their continued support throughout life...

Chapter 1

INTRODUCTION

Over the past half-century, the use of digital systems to perform tasks has allowed many people to increase their efficiency and productivity in various endeavors. In fact, the use of digital devices has become prevalent in almost all avenues of everyday life, from the workplace to the home. As such, much investment has been placed in the techniques and methodologies used in the design and production of these systems. Many processes have been developed to allow designers to quickly produce systems that not only perform a task correctly, but efficiently and cost effectively as well.

One technique is to divide a task to be implemented in a digital system into various subtasks that as a whole complete a job. Indeed, a digital system frequently consists of subsystems that transmit data with one another to perform a task. This interdependency of data between subsystems must be coordinated in such a way that the functionality of the system conforms to specifications.

1.1. Synchronous Design Methodology

One method to coordinate data transmission between subsystems is to use a *synchronous design methodology*. A digital system developed with a synchronous methodology is one that allows data movement between components only at specified instances of time. The usual way of accomplishing this is to have a common timing reference between all components that dictates when data may be transmitted. Such a timing reference can be implemented as a clock signal. In essence, a component will assume valid data is present at its' inputs during a time specified by a clock signal and will use the input data based on this assumption.

1.1.1. Global Clock Signal

Suppose we have a system in a state where all components have valid inputs, but have not started computation. If all components start computation at the same time, one may say that all components will be done by the time the slowest component is done; let this slowest time be called t. We can safely assume that the next cycle of computation may start at time $t+\triangle$, where $\triangle$ is some positive time, if this t does not change through the entire execution of the system. This is known as a *bounded delay* or *worst case delay* assumption. Now, rather than having each component be aware of such a t and keeping track of time internally, a *global clock signal* can be connected to all components that can tell the components when they may assume valid input data have arrived. This is a simple solution to what could be a hard problem using other techniques. To further simplify matters, positive-edge triggered elements are used to specify how a clock may dictate when computation may proceed. The global clock is the predominant technique for creating synchronous digital designs, so much so that synchronous systems and edge triggered, globally clocked systems are synonymous.

1.1.2. Synchronous Design Issues

As technology allows for more complex systems to be implemented, setbacks have been encountered when designing systems using global clocks. One assumption needed for a global clock signal to be feasible is a uniform delay clock distribution (i.e. all components receive the global clock signal at essentially the same time). As clock speeds increase, the physical properties of the wires that this signal propagates over must be taken into account. However, in current synthesis techniques, especially at higher levels of abstraction, wire delay is typically not a primary concern. It is only during the later stages of the design cycle that the physical topology of wires, and the associated delay, may be truly assessed. The overall effect of this neglect of wire delay is a potential cause for *clock skew* that can adversely effect correct operation of the circuit.

Clock skew occurs when all components connected to the clock signal do not receive transitions of the clock signal at the same time. An adverse consequence of clock skew is that all the memory elements connected to the clock signal do not latch data at the same time. This may lead to incorrect computation. Design time needed to correct such cases has become a large part of the design cycle, and is commonly referred to as *timing closure* or *timing convergence*. Various factors have made clock skew prevalent, not limited to increased component count, decreasing feature sizes, and faster clock requirements.

To compensate for clock skew, there are various techniques to account for clock skew. Downgrading the clock speed may solve clock skew, but does not allow for optimal performance. Increasing the drive strength of the clock generator of increasing the conductance of the distribution network in the chip to make the signal propagate faster is another option, but the circuitry used to generate the signal uses more power and chip area and generates more heat. Often times, the network is over designed to account for clock skew to meet time-to-market constraints. There are additional consequences that arise when the clock distribution network is made more powerful. There may be times when certain circuits in a design will not perform any meaningful computation. However, these components are still connected to the clock, which must always run. This leads to increased power consumption and heat generation. Although not a primary concern in the past, managing power dissipation has become very important in the emerging portable, hand-held device market.

Another method is to use different memory elements, rather than traditional edge triggered devices, which can allow designs to better handle clock skew. Domino logic techniques have been used to speed up the combinational logic portion of circuit, which may be 1.5 to 2.0 times faster than a static CMOS counterpart [12]. Domino logic based designs may be skew-tolerant given that certain rules are followed, but these types of circuits tend to be more complex than standard synchronous designs.

1.2. Asynchronous Design Methodology

Another design methodology is the *asynchronous methodology*. Commonly, any design that does not have an explicit clock signal is referred to as an asynchronous design [22, 34]. This methodology has not been widely embraced by designers mainly because implementing a design using the methodology has traditionally been difficult. However, as the problems of clocked designs grow and the techniques for implementing asynchronous designs mature, the asynchronous methodology has garnered renewed interest.

1.2.1. Survey of Asynchronous Design Models

Over the years many approaches for designing asynchronous devices have been developed. In the early days of asynchronous design, the main contributions were due to Huffman [14] for his work involving bounded-delay models that are named after him. These Huffman circuits have been used in practical settings; however, in more recent endeavors, the concept of an unbounded delay model has taken precedence. Such circuits are usually termed as self-timed, delay-insensitive, or other various names.

1.2.1.1. Bounded-Delay Models

One of the first asynchronous models, called *bounded-delay* or *Huffman* models, uses a similar assumption as synchronous designs. Namely, the delay through gates and wires are assumed to be known and finite. If one thinks of a digital system as a *finite state machine* (FSM), one can see how a system goes from state to state based on the current state and possible inputs. A clocked system uses the clock signal to dictate when a next state may be entered based on the current state, whereas an asynchronous bounded-delay circuit uses explicit delay circuitry to achieve a similar result. The discrete nature of a clocked system hides unwanted transient signal behavior on the current state and input signals and only uses stable signals to compute the next

state. However, in an asynchronous bounded-delay circuit, unstable behavior may cause unwanted states to be entered. Unstable signals that cause such behavior are called *hazards*. Therefore, an attempt must be made to ensure that hazards do not appear.

There are various methods in Huffman circuit design that do this. One method dictates that only one input signal at a time may change. This is known as the *fundamental mode* of operation. There are extensions to the methodology that allow for non-fundamental modes and burst modes that allow for certain sets of inputs to toggle at the same time. However, the constraint of only changing certain inputs at a time implies that going from state to state may take longer than a synchronous circuit. Additionally, buses are impractical in such methods because bus lines all frequently toggle at the same time. Another factor is a phenomenon called *additive skew* [13]. This skew comes about when bounded-delay circuits are placed in series and the overall affect is that the throughput of the system goes detrimentally down.

1.2.1.2. Delay Insensitive Models

Unbounded-delay models, also called *delay insensitive* (DI) models, assume that the delays on gates or wires are unbounded. Meaning that a component may take any amount of time to complete a computation and wires may take any amount of time to propagate a signal. Due to the unbounded nature of timing, this approach implies a mechanism is needed to indicate when inputs are available. The predominate method is to setup a handshaking protocol between communicating components. Other methods include forms of completion sensing, such as *Current-Sensing Completion Detection* (CSCD) [6]. Handshaking protocols with a direct relationship with the work presented in this thesis will be discussed in the next chapter.

Other similar models are the *speed-independent* (SI) and *quasi-delay-insensitive* (QSI) models. SI makes the assumption of unbounded gates but has wires with negligible wire delay. QSI has both unbounded gate and wire delays but includes a

concept called *isochronic forks*. An isochronic fork is a forking of wires where the delay in each path is essentially identical. It is known that isochronic forks are not easily implemented as the assumption of equal delay down each branch necessarily dictates that the devices the individual branches connect to also have roughly the same threshold voltage [19, 1]. This characteristic is not easily controllable in arbitrary circuits.

1.2.1.3. Hybrid Models

Models that use components with both synchronous and asynchronous characteristics are being investigated as well. *System-on-Chip* (SOC) devices may use synchronous IP blocks and asynchronous inter-block communication protocols. Mixed Asynchronous Synchronous Systems (MASS) have been investigated in [35]. Additionally, Globally Asynchronous, Locally Synchronous (GALS) methods have been shown to have promise [20].

1.2.2. Asynchronous Design Characteristics

In the literature [13], potential benefits of some asynchronous designs include:

- No Clock Skew - by definition, since there is no clock, there can not be clock skew. However, other issues arise when the simplifying mechanism of a clock is removed.
- Lower Power - as described before, in a synchronous circuit, portions may be activated that do not perform meaningful work. Asynchronous circuitry tends to limit circuit activation to portions involved in the current computation. Additionally, the behavior of self-timed designs may make idle modes easier to implement [24].
- *Reduced Electro-Magnetic Interference* (EMI) - Without a clock, it has been shown that the overall EMI characteristics of a system are lowered in self-timed devices. Additionally, in emerging technologies where security is a concern such

as smartcards, lower EMI makes it harder for devices to "eavesdrop" on the operation of such a device [21].

- Average-case instead of worst-case performance - Cells in an asynchronous design can allow components to process data as soon as it is ready, rather than having to wait for the slowest component to finish.

- Automatic adaptation to environmental/physical properties - physical factors such as fabrication variations and the physical environment may affect the performance of circuitry. Synchronous circuitry must assume the worst of these factors and change the clock accordingly. Delay insensitive circuitry may avoid this problem because of data completion mechanisms, and will run as fast as the environmental/physical factors will allow.

Even with the problems associated with clocked circuitry and the benefits of asynchronous design, many designers have little desire to change methodologies. Although many have expressed concern over theoretical physical limitations of synchronous designs, innovation has allowed such designs to work properly using cutting edge processes and higher specification requirements. Although there has been much investigation into asynchronous design, and some fairly large circuitry implemented, the gains due to asynchronous design are often considered to be not worth the effort required by the designer. Moreover, the design flow for synchronous designs is known and time tested. CAD tools for synchronous designs are mature and robust, while CAD tools for asynchronous methodologies are typically in prototype form and may be hard to use. Conventional circuit designers are reluctant to attempt to implement asynchronous designs if the learning curve and usability of the CAD tools is high.

1.3. Phased Logic

As stated above, a major factor for the dismissal of asynchronous design methodologies is that they are hard to use. Additionally, automation tools for implementing such designs require extensive knowledge of the underlying processes. These facts are

a motivating factor for *Phased Logic* (PL). In [16], Linder devised an asynchronous design methodology intended to aid the traditional synchronous circuit designer in formulating self-timed designs. Phased Logic allows a designer to use existing synchronous design tools such as hardware description languages at the RTL level to create a system with a global clock and then to convert this design from a clocked design to a clock-less, self-timed design. This methodology allows a designer to more readily build on conventional tools and knowledge rather than having to abandon conventional approaches.

1.4. Impact and Contributions of this Research

In this work, mechanisms for increasing the overall performance of PL circuitry [26], namely *Early Evaluation* (EE) and *slack matching buffering*, are discussed. The main contribution of this thesis is the derivation of automatic techniques for the implementation of these performance enhancement mechanisms.

EE allows a PL cell to produce an output based on only a subset of available input signals. This allows a gate to evaluate earlier than it normally would since it does not have to wait for all input signals to arrive. Since a key motivation for using asynchronous designs is that it operates using average case timing, being able to lower the average case time is beneficial.

Slack matching buffering, previously called *Bit-Level Dataflow* [26], is the process of inserting buffers in certain data paths to improve the throughput of a system. This is due to the ability of such buffers to balance interconnected pipelines, which minimizes stall situations when pipelines are waiting on one another.

Finally, optimization techniques such as EE and slack matching change the flow of data tokens within a PL circuit. Being able to visualize these changes allows a researcher to better understand how these techniques affect a circuit on a global level. A visualization tool that allows this, called `PLFire`, is developed as part of this research and is discussed.

1.5. Organization

The remainder of this thesis is organized as follows. Chapter 2 provides a review of the major topics regarding Phased Logic. Chapter 3 gives information relating to the formulation of a speed-up mechanism called Early Evaluation. Additionally, methods for determining a trigger function, which is a major component of Early Evaluation, will be discussed. Chapter 4 presents implementations of the methods described in Chapter 3 as well as experimental results. Chapter 5 details another optimization technique called slack buffer matching and provides experimental results. Chapter 6 presents a visualization tool that can be used to view the token flow of a PL system. Lastly, Chapter 7 concludes the thesis and includes final observations and ideas for further research.

Chapter 2

PHASED LOGIC OVERVIEW

Phased Logic (PL) is a self-timed design methodology that eliminates the need for a global clock and allows for automated mapping of a clocked netlist to a self-timed netlist. The technique is first proposed by Linder in [16], and subsequent work has been documented in [17, 26]. The basic difference between conventional clocked designs and Phased Logic is the use of two-phase data conventions [31, 7], micropipelines [33], and marked graph theory [5]. Additionally, two types of PL architectures have been studied that are better suited to FPGA and ASIC implementations respectively. To aid the designer, a design flow has been established that closely follows traditional approaches.

2.1. Marked Graph Theory

Marked graphs [5], a specific type of *Petri Net* [23], is used as a basis for the automated synchronous to PL-style asynchronous mapping technique. Such graphs are known to model resources flowing through a network and one may naturally describe a digital system in the same way.

Let G be a *directed graph* composed of two types of nodes: *transitions*, T, and *places*, P. Places must have a $degree_{in}$ and $degree_{out}$ of 1 and transitions must also have $degree_{in}, degree_{out} > 0$. Nodes adjacent to transitions must only be places, and nodes adjacent to places must be only transitions, in other words a closed, strongly connected 2-chromatic graph with T's of one color and P's the other.

Let M be a token mapping of G. A token mapping is a set of pairs $\langle P, t = \{0, 1\}\rangle$. If $t = 0$, then P does not have a token, else P has a token. Let M_I denote an initial

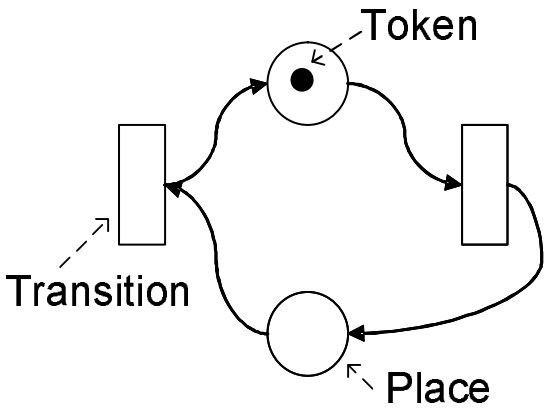


Figure 2.1. Marked Graph Example

token mapping of G. Places are allowed only to have 0 or 1 tokens. This set $\{G, M_I\}$ is potentially a marked graph. Other constraints needed for this structure to be a marked graph will be discussed shortly.

A marked graph also has a notion of transition firings. A transition fires when all place inputs have a token. Upon firing, the transition will remove the tokens from the input places, and insert tokens on the output places. One may imagine how tokens may flow through a graph based on this firing rule.

To ensure a firing does not cause a place to have more than 1 token, two conditions must be met: *liveness* and *safety*. Liveness occurs when all directed circuits within the graph have at least one token. Safety is met when all places are part of at least one directed circuit with exactly one token. So a $\{G, M_I\}$ that is both live and safe is a marked graph. Note that in Figure 2.2 that cycle a is unsafe and cycle b is not live. In a similar graph shown in Figure 2.2, cycles a' and b' are both live and safe.

To create a circuit that has the same behavior of a marked graph, various concepts and circuits need to be introduced. Concepts that lead to how a circuit may represent tokens and the firing behavior of a marked graph will be discussed next.

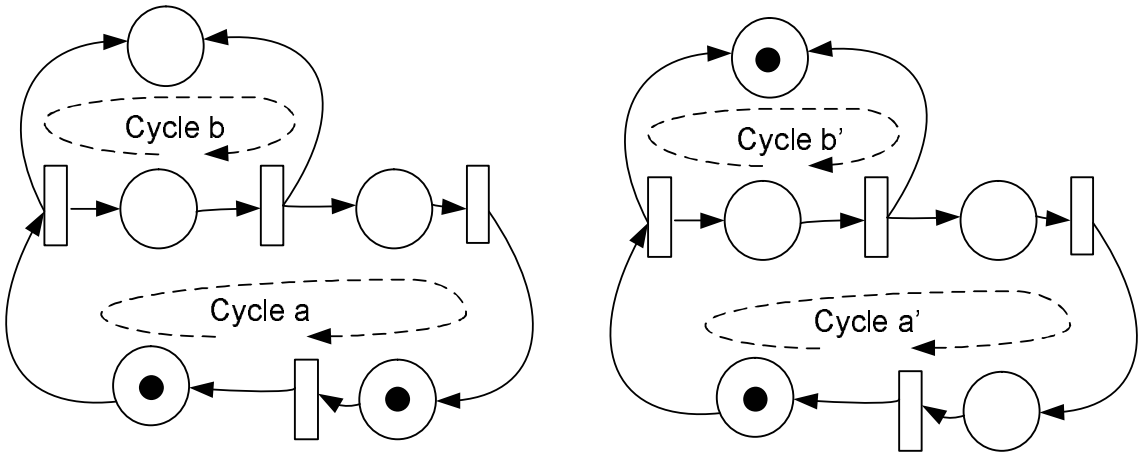


Figure 2.2. Safety and Liveness in Marked Graphs

2.2. Transition Signaling

In a traditional synchronous design, it is usual for one clock edge type to indicate an event. Transition signaling merely refers to the concept that either edge, rising of falling, may indicate an event. This concept is needed in various types of communication protocols used in PL (i.e. the way tokens are propagated) . Additionally, a benefit of such a scheme is that there is no return to a neutral state in the clock before another event may occur. This may save on time and energy costs associated with a neutral state transition [33].

2.3. Two-Phase Bundled Data Convention

The two-phase bundled data convention [31] is a local communication mechanism for a sender and receiver to communicate with one another. Figure 2.3 shows the basic structure of a sender and receiver using the convention.

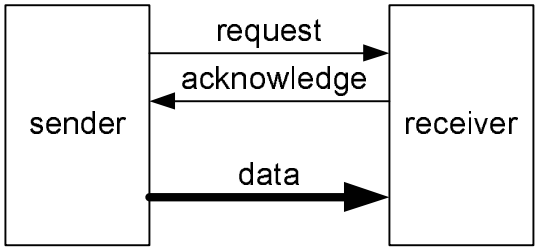


Figure 2.3. Sender and Receiver in a Two-Phase Bundled Data Configuration

The communication protocol works as following:

1. Sender places data on the data bus
2. Sender produces an event on the request line to indicate valid data is available
3. Receiver takes the data
4. Receiver produces an event on the acknowledge line to indicate data has been accepted and new data may be applied to the data bus

Note that the data and request line are usually treated as one entity, or bundle, however the request signal must reach the receiver *after* the data does in order for the protocol to work. The acknowledge and request signals use transition signaling to indicate events on those lines. This type of protocol is used in *coarse-grained PL*, which will be discussed shortly.

2.4. Level Encoded Dual Rail Signaling

Level encoded dual rail signaling (LEDR) [7] is a two-wire data signaling scheme whose characteristics are shown in Figure 2.4. Note only one bit in the dual rail will change between any valid code word. Moreover, for a particular computation, a signal changes phase and can also change in value; the concept of even/odd phase will be

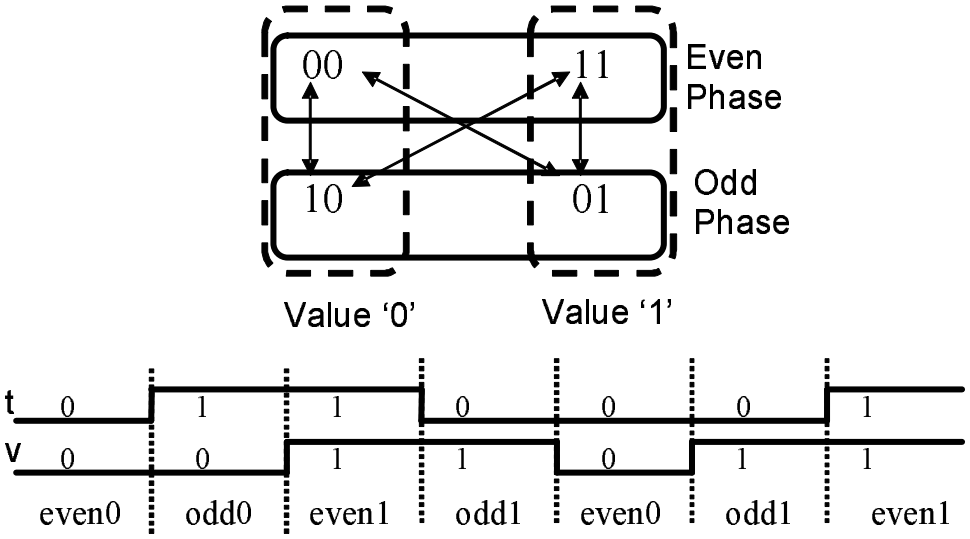


Figure 2.4. LEDR characteristics

discussed in the next section. This type of protocol is used in *fine-grained PL*, which will be discussed shortly.

2.5. Even/Odd Phases and Firing Convention

As we are implementing a marked graph in circuitry, the abstract notion of a token must be implementable. A notion of even and odd phase, in conjunction with transition signaling, may fully describe the behavior of a token without resorting to request and acknowledge signals between all connected components.

First, a definition on how we treat phase to indicate a token on a particular gate is presented. Allow a gate to have an associated phase, even or odd. A gate may only modify its outputs, and an output signals' phase is always the opposite of the driving gate. A gate is not capable of changing the phase of its inputs but is able to determine its' input signals' individual phases. If an input signal phase and the gate phase are the same, then there is a token associated with that input present.

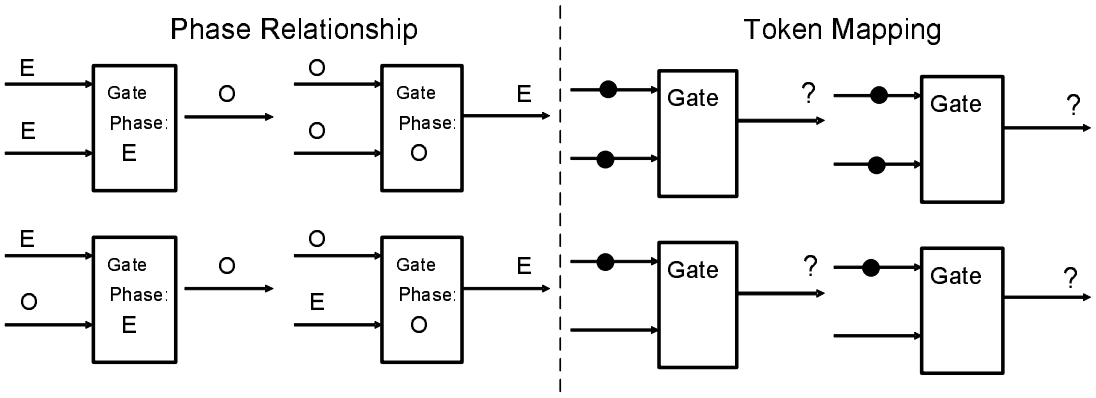


Figure 2.5. Phase and Token Relationship

To extend the notion, suppose we have a marked graph G. Recall that a transition may represent a gate and places may represent signals, and this notion will be used interchangeably in this section. Now, instead of tokens, we will use phase to indicate a marked graphs' initial token mapping. Suppose all gates have odd phase. This means all places must have even phase as well. Now suppose, we toggle one of the gates to even phase. This necessarily means that all output signals of this gate have odd phase. This will then indicate a token is on the inputs that this gate drives. Note that if the initial token mapping does not indicate a token is needed, we may invert the input of a gate being driven. It can be seen that this process uses a phase convention to fully specify tokens in a marked graph.

A notion of token firing is also present in a marked graph. By definition, there is at least one transition that is ready to fire giving a particular token mapping since we are considering line graphs only. Now consider such a marked graph with the concept of phase used to denote tokens. This graph contains some gate with the proper gate/input phase relationships indicating it is ready to fire. When a gate fires, it will change its internal phase and the phase of its' output signals. This happens throughout the system and allows tokens to flow through the indeced marked graph.

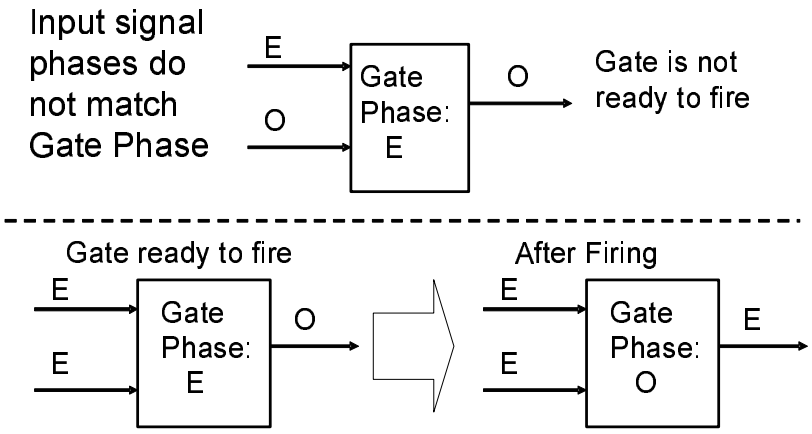


Figure 2.6. Phase and Token Firing Rule Relationship

2.6. Micropipelines

Micropipelines, introduced by Sutherland in [33], are a simple form of an event-driven elastic pipeline which can serve as a framework for pipelined processors. Micropipelines use transition signaling, the bundled data convention, and event driven logic elements as the basis of the approach. Event driven logic computes an output only when a particular event on the inputs occur. In particular, a *C-Muller gate* is the AND operation with respect to events [30]. When all inputs transition to the same value then the output transitions to this value, else the output retains its current value. This means that it will only start computation when all inputs have had a transition event occur. A benefit of such an approach is that it is a modular approach, where one can easily plug in components and have a functional circuit without having to worry about delay between components. Figure 2.7 presents an illustration of a micropipeline structure as shown in [33].

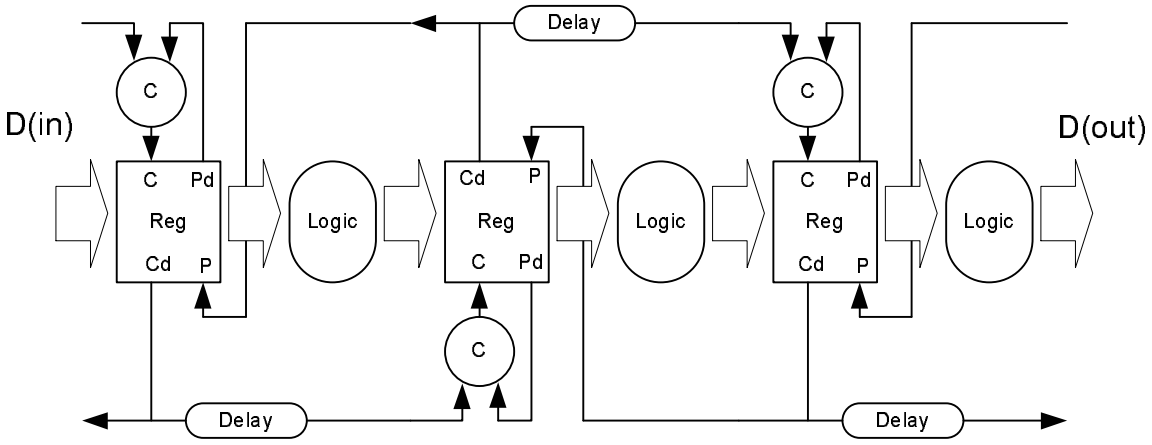


Figure 2.7. Basic Micropipeline Configuration

2.7. Fine-Grained and Coarse-Grained Approaches

There are two approaches considered in applying PL techniques: the fined-grained [28] and coarse grained [27] approaches. The fine-grained approach assumes that programmable PL cells are the basic block of the system and is thus suited for FPGA-type architectures. A a fine-grained PL cell has a look-up table programmed to perform a function and phase control circuitry to handle data flow. The process of transforming a clocked netlist to a PL netlist occurs at the gate level. Each gate in the clocked netlist is replaced with a PL cell and additional feedbacks are added to ensure liveness and safety. Additionally, LEDR signaling is used to enable token flow.

The coarse-grained approach is better suited for ASIC designs. Where the fine-grained mapping process works at the gate level, the coarse-grained approach allows for mapping to occur at the module level. Modules are collections of standard cells, such as an adder, with PL control logic inserted around the module. This approach allows for already developed standard cell libraries to be used in a PL design. In the coarse-grained approach, the two-phase data bundle convention is used to enable token flow.

2.8. PL Mapping Process

Now that the components of a PL system have been described, we may discuss how a clocked design is converted to a self-timed design. Note that although the basic blocks presented are important to PL, the designer does not need to know anything about them. The PL mapping process is automated and can transform a netlist of a clocked system into a PL system composed of the various building blocks described previously. The basic steps for converting a clocked netlist to a PL netlist include transforming gates or collection of gates (partitions) into PL equivalents. This is merely wrapping PL control logic around these components. Additionally, the interconnecting wires are converted to two-phase versions so that tokens may be represented. Note that fine-grained uses LEDR and coarse-grained uses two-phase bundled data. Finally, single rail feedback signals and tokens are added to the netlist to ensure liveness and safety.

2.8.1. Fine-Grained Component Transformation

In the fine-grained approach, we implement a clocked netlist in an FPGA type architecture. As such, the netlist we are provided is composed of *look-up tables* (LUT) and *flip-flops* (DFF). The process used to transform the clocked netlist into a PL netlist is to replace each component with a PL equivalent. An equivalent PL component is merely the component with PL control wrapper circuitry. To ease implementation in a FPGA, such an FPGA would need to provide such PL structures. We refer to PL compute gates (the transformed LUTs) as *through gates* and PL sequential elements (the transformed DFFs) as *barrier gates*. This approach is not limited to LUTs and may be used to transform other types of gates in similar manner.

2.8.2. Coarse-Grained Component Transformation

In the coarse-grained approach, we can deal with collections of components and treat them as single modules; for instance, in [27], a netlist of LUTs and DFFs are

partitioned into various modules. One may wrap PL control logic around such a partition and treat it as a single entity in a PL netlist. Additionally, in clocked ASIC designs, standard cells are used. This may be thought of as a partition, and we may wrap PL control logic around such a cell in a similar manner. If an output of a standard cell or a partition is registered in the clocked netlist, then the output is also connected to a barrier gate in the PL netlist.

2.8.3. Feedback Insertion Rules

The last step in the mapping process is the addition of feedback signals to ensure liveness and safety. These feedbacks signals may be single rail since they only propagate timing information, no data. The feedback generator assumes all barrier gate outputs have initial tokens. Rules to ensure that safety is met given this assumption include the following:

- Loops used to ensure safety are not allowed to pass through more than one barrier region as this would place two tokens on the loop. If necessary, a buffer function *splitter gate* may be inserted in between barrier gates to provide a source and sink for feedback.
- Feedbacks that originate from barrier regions have tokens.
- Feedbacks that originate and terminate between through gates must have a token in order for the loop to be live.
- Feedbacks are not allowed to originate from a barrier gate and terminate on a barrier gate as that would compromise safety.

A clocked circuit has a concept of storage and combinational subcircuits. In PL, a storage subcircuits are referred to as *barrier* gates and combinational gates as *through* gates. Sequential regions are usually just a collection of *flip-flops* (DFF) and combinational regions a collection of combinational gates; barrier regions and through regions have a similar definition with respect to the PL regions.

The first stage in a PL mapper is to identify sequential regions and combinational regions. One approach is to merely identify all DFFs as barrier regions and all combinational gates with through regions. This is done for fine-grained designs.

The next stage is to warp each region in appropriate PL control circuitry. This wrapping of PL control logic is where one starts referring to sequential regions as barrier regions and similarly combinational regions as through regions. This is done using a standard template which allows the regions to be plugged into the template with minimal effort. In the fine-grained approach, since we a converting individual gates, we say barrier and through gates instead of region. All signals are also converted to two-phased bundles or LEDR depending on the approach used.

2.9. PL Design Flow

A generic PL design flow is shown in Figure 2.8; flows geared towards fine- and coarse-grained approaches may be found in [27, 28]. As input, a clocked netlist is given to the structure analyzer. In this block, techniques may be used to optimize a clock design to utilize methods available in PL. This block outputs a restructured netlist that is optimized for PL and has the PLMapper convert this netlist into a PL system. Note that the design flow is only for converting a clocked design to a PL design, and may easily fit into the traditional design cycle for sequential systems.

2.10. Summary

In this chapter a presentation of the PL design flow was described. Details of the basic blocks needed for a PL circuit as well as the methods used for converting a sequential design into a PL design were described. In particular, the structure analysis block of the PL design flow applies directly to this thesis. In the following chapters we discuss methods for analyzing portions of circuitry to determine whether early evaluation and slack matching may occur.

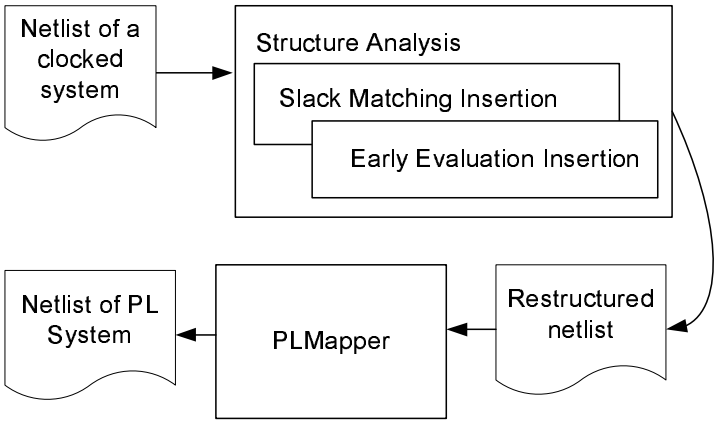


Figure 2.8. PL Design Flow

Chapter 3

EARLY EVALUATION THEORY

Self-timed designs perform using an average case timing characteristic rather than worst case timing based on the slowest component. Although, theoretically, self-timed designs should operate at increased speeds as compared to synchronous designs, this may not be observed in a practical implementation. One reason for this is that the overhead needed for local communication between components can be greater than any benefit gained from an average time gate firing. A method called *Early Evaluation* (EE) can allow an asynchronous circuit to outperform a synchronous circuit by allowing its components to perform computation only on a subset of available data inputs. Depending on the incoming data, EE allows for a systems' average case timing to be lowered, potentially overcoming the communication overhead.

3.1. EE Derivation

As opposed to synchronous designs, the concept of input signal arrival time and input availability is present in PL designs. In synchronous designs, a combinational circuit is separated from new inputs by memory elements, which act as a barrier differentiating "current" and "next" input values. However, in a self-timed system this sort of barrier is unnecessary. A combinational circuit may have access to input values even when all inputs values are not yet available. It will be shown that with respect to such an available set, the other variables may be determined to be inessential and it may be possible to proceed with computation.

3.1.1. Basic Terminology

Let $\mathcal{B} = \{0, 1\}$. An n-input Boolean function f is defined as $f : \mathcal{B}^n \rightarrow \mathcal{B}$. An n-input vector $\underline{B} = \langle b_{n-1}, b_{n-2}, ..., b_0 \rangle$ of f is an ordered set of Boolean variables; were $|\underline{B}| = n$ denotes the size of $\underline{B}$. An assignment to $\underline{B}$ is denoted as $[\underline{B}]$; $\underline{B}$ has a total of 2^n such assignments. A given $[\underline{B}]$ such that $f([\underline{B}]) = 1$ is denoted as minterm of f. Let $\#min(f)$ denote the number of minterms of f. A *cube* is a set of minterms whose intersection results in a single product that covers all minterms in the set. A cube is said to be a *cover* of these minterms. A j-cube is cube with $(n - j)$ literals, thus a minterm is also a 0-cube. The j variables that have been removed in a particular cube are called *don't cares*.

3.1.2. Inessential Variables Property

For a given n-variable Boolean function, there are 2^n possible input vectors. Yet, there are only 2 possible results that this function can have for each input vector: 0 or 1. With such a small range, one may intuitively guess that for a particular variable assignment, the output of a function may not greatly depend on all of the variables. This is not to say that we are dealing with vacuous variables all of the time, but that for some variable assignments, some variables may not be needed to determine the result of the function. Variables that are not needed when a subset of variables are given an assignment are termed the *inessential variable set for the assignment*. An elementary example portraying the inessential property using the real-valued functions f follows:

$$f(x, y, z) = z(x + y)$$

$$\text{When } z = 0, f(x, y, z) = 0, \ \forall \{x, y\}$$

In this case, x and y are inessential when z equals 0. In a real-valued setting, the opportunity for application of this property may be trivially small, but in binary valued Boolean logic the only two variable values are 0 and 1. This inessential property is prevalent since 0 is the additive identity and 1 the multiplicative identity in $GF(2)$.

Being able to determine the value of a function without needing all of the input variable assignments is the main property that EE exploits. Note that $\{x, y\}$ is not an inessential set when $z = 1$.

3.1.3. Available Variables Property

Normally, a gate can compute only when all of the inputs have been assigned with respect to the current computation cycle. In terms of PL, computation may occur when all inputs have the same phase as the PL gate. Yet, an n-input PL gate may have $[0, n-1]$ inputs assigned. The set of inputs that have been assigned with respect to the current cycle is known as the *available set*. This of course gives rise to another set, the *unavailable set*.

If it could be determined whether the unavailable set is inessential with respect to the available set, then the PL gate can compute and attain the same result as if all the inputs were available. If we go back to the real-valued example, if z is known to be 0 before x and y are available, we can automatically say that the output should be 0. This can be advantageous if $\tau([z] = 0) < \tau([x]), \tau([y]) \Rightarrow f = 0$, where $\tau(p)$ indicates the time when proposition p is true.

3.1.4. Early Evaluation Property

For an arbitrary n-input PL gate at a time t that is between firings we may divide the available and unavailable inputs at time t into two vectors, denoted as $\underline{A}$ and $\underline{U}$ respectively, where $|\underline{A}| = k$ and $|\underline{U}| = n - k$. Note there are 2^k possible assignment vectors for $\underline{A}$; let $[\underline{A}]_j$ denote the j^{th} such assignment. If $\underline{U}$ is determined to be inessential with respect to $[\underline{A}]_j$ then the PL gate may fire, since the output is only dependent on the assignment $\underline{A}$. If $\underline{U}$ is inessential with respect to $[\underline{A}]_j$, then $[\underline{A}]_j$ is said to give rise to an early evaluation opportunity. A function that has any $[\underline{A}]_j$ that gives rise to the early evaluation property is said to have an early evaluation property.

3.1.5. Existence of Early Evaluation Property

The set of unavailable inputs, $\underline{U}$, must be found to be inessential with respect to $[\underline{A}]_j$ in order for EE to occur. The domain of an n-input Boolean function f consists of 2^n input vectors. Each of these elements in conjunction with f maps uniquely to either 0 or 1. Now, suppose we have a set of available inputs, $\underline{A}$, where $|\underline{A}| = k$, with the assignment $[\underline{A}]_j$. We can then group the 2^{n-k} elements of the domain of f that correspond to $[\underline{A}]_j$. If all 2^{n-k} elements map to the same value, then $[\underline{A}]_j$ fully determines the result of the function with respect to the partial assignment. One may easily construct examples that show this property by assuming some assignment and then appropriately setting the mapping. Note an n-input Boolean function that is a tautology is a degenerate case of this property.

In terms of minterms and cubes, each $[\underline{A}]_j$ that can early evaluate may be thought of as a set $(n-k)$ cubes with the same support set of the function or its complement. Therefore, if a function or its complement has a cube that is not a minterm then the function has the early evaluation property. Furthermore, this implies that the number of prime implicants of a function with the EE property is less then $2^n - 1$. The only functions that do not have less than $2^n - 1$ prime implicants are the XOR and XNOR functions. Therefore, the XOR and XNOR functions are the only functions that can not be early evaluated.

3.2. Trigger Functions

Handling all possible $\underline{A}$s for a particular logic function would be cumbersome in implementation. As such, one may "hard-wire" a unique $\underline{A}$ to be used to identify EE opportunities. A Boolean function, known as a *trigger function*, uses an assignment on a unique $\underline{A}$ to determine if $\underline{U}$ is inessential. To accommodate EE, the structure of a PL cell must be modified to contain a master and trigger function. To accomplish this two standard PL Cells are connected together with additional logic. One may refer to [36] for further details on the implementation of EE in PL.

3.2.1. Definition

Let M be an n-input Boolean function, denoted as the *master function*. The role of the t-input Boolean function, T, is to indicate whether EE is possible given the assignment of a subset of inputs of M. If EE is possible then T evaluates to 1. We call such a function a *trigger function*. Formally, we have the following:

Let $\underline{X} = \langle x_{n-1}, x_{n-2}, ..., x_0 \rangle$ be an ordered set of Boolean variables and $M(\underline{X})$ be a Boolean function with support set $\underline{X}$. Let $\underline{\tilde{X}} = \left\langle x_{t_{k-1}}, x_{t_{k-2}}, ..., x_{t_0} \right\rangle$ be an ordered subset of $\underline{X}$. A trigger function, $T\left(\underline{\tilde{X}}\right)$ of $M(\underline{X})$ is defined as

$$T\left(\left[\underline{\tilde{X}}\right]\right) = \begin{cases} 1, & M_{\left[\underline{\tilde{X}}\right]}(\underline{X}) = 1 \text{ or } \overline{M}_{\left[\underline{\tilde{X}}\right]}(\underline{X}) = 1 \\ 0, & otherwise \end{cases}$$

where $\left[\underline{\tilde{X}}\right]$ indicates an assignment of the variables in $\underline{\tilde{X}}$ and $M_{\left[\underline{\tilde{X}}\right]}(\underline{X})$ is the cofactor of M with respect to $\left[\underline{\tilde{X}}\right]$. Note that we explicitly are making $\underline{A} = \underline{\tilde{X}}$ and $\underline{U} = \underline{X} - \underline{\tilde{X}}$. In terms of cubes, we may specify a particular trigger function as the cubes of the onset of either M or $\overline{M}$ that have at the variables in $\underline{U}$ as don't cares.

3.2.2. Coverage

An n-input Boolean function, M, has 2^n different input vectors. We wish to have a notion of how often a trigger allows for an EE opportunity of its master function. From the definition, a minterm of T corresponds to a set of disjoint $(n-t)$-cube in either M or $\overline{M}$. The maximum number of minterms in $M \cup \overline{M} = 2^{|n|}$. We define coverage of a t-input trigger T with respect to n-input master M as

$$Coverage(T) = \frac{\#min(T) \cdot 2^{n-t}}{2^n} = \#min(T) \cdot 2^{-t}$$

3.3. Trigger Function Identification Schemes

In the definition of a trigger function, it is stated that $\underline{A}$ is explicitly forced to a particular $\underline{\tilde{X}}$, which is then used to form T. This implies that a suitable $\underline{\tilde{X}}$ must be found. In the schemes to follow, we assume we have found a desirable $\underline{\tilde{X}}$, and find the associated trigger function.

a	b	c	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)

a	b	c	M	T
0	0	0	0	**1**
0	0	1	0	
0	1	0	0	**0**
0	1	1	1	
1	0	0	0	**0**
1	0	1	1	
1	1	0	1	**1**
1	1	1	1	

(b)

Figure 3.1. Truth Table Example

3.3.1. Truth Table Approach

A Boolean function may be represented using a truth table, which lists both the domain and range of a function in a convenient way. One may use this structure to determine whether a potential available set has a corresponding trigger function.

Suppose we have the function $M = ab + ac + bc$. A corresponding truth table is shown in Figure 3.1a.

Now, suppose we wish to find a trigger function whose support set is $\{a, b\}$; in other words, $\underline{A} = \{a, b\}$, $\underline{U} = \{c\}$. Therefore, we need to look at the cubes of f using a subset of variables from $\{a, b\}$. We can do this via a truth table by just marking out the c column, as shown in Figure 3.1b. There will be rows with the same a, b values, which correspond to an assignment of $\underline{A}$. The rows with the same value for a, b may be grouped together and if they all resolve to the same output that assignment is used to set T to 1, else it is a 0. This process gives $T = ab + \overline{a}\overline{b}$. One may intuitively determine the general algorithm, and its description will be held off until the next chapter.

In order for this to work, we must be able to hold a truth table in memory. Since the size of a truth is exponential with respect to n, the space complexity of the algorithm is exponential. Additionally, although we group the minterms, we still must check them all. Therefore, the algorithm has an exponential time complexity as well.

A less exact approach would be to minimize a M and $\overline{M}$ and then group cubes that have similar don't care sets. These sets by definition are trigger functions for M. It is less exact because a function may be represented with more than one minimized cube lists; however, such a trade-off is reasonable considering the computation time and space saved.

3.3.2. BDD Approach

The truth table approach is suitable for finding trigger functions of small size. However, for functions with large variable support sets, the space and time requirements of a truth table based algorithm are too large. One may notice that the algorithm works closely with the representation of the logic function and fortunately there are more compact ways of representing a function. This observation leads to an algorithm based on a *binary decision diagram* (BDD) for trigger identification.

3.3.2.1. Binary Decision Diagrams

A *binary decision diagram* (BDD) is a method for representing Boolean logic functions. A BDD may be visualized as a binary tree structure where each path down the tree represents a cube of the logic function. If the cube evaluates to 1, then the path terminates at a 1-node, else a 0-node. There are various types of BDDs but the one to be focused on is the *reduced ordered BDD* (ROBDD). The ROBDD is so often used that we will interpret the usage of the abbreviation BDD to denote ROBDD unless otherwise specified. An ROBDD imposes the following rules on the structure of the tree:

- Each level of the tree is associated with one unique variable in the support set of the function
- The left and right subtrees of a node cannot be identical
- There cannot be identical subtrees in the ROBDD

ROBDDs are canonical representations for Boolean functions; further reference on the subject may be found in [3].

Some notation used to refer to elements in a BDD are the following: Let N be a BDD and $n \in N$ be some node in the ROBDD. $N.root$ is the root node of the BDD. $n.0$ and $n.1$ refer to the left and right children of n. n_0 and n_1 refer to the edges between a parent and the left/zero and right/one child nodes.

3.3.2.2. BDD based Trigger Function Extraction

An algorithm for transforming a ROBDD into a trigger function is shown in Figure 3.2. It is not desirable for the master to be transformed into a trigger, one would rather construct a trigger without losing the master. This transformation is presented as it gives a clearer idea on how one gets a trigger from a master function through the use of a BDD. The implementation to be described in the next chapter will illustrate an algorithm were the master function is not destroyed. Of course, one could merely make a copy of the master and then perform the transformation, but this uses more space and is not necessary.

BDD_TransformTrigger(N **,** $\underline{A}$ **,** $\underline{U}$ **)**
{
Input: BDD N , $\underline{A}$, and $\underline{U}$
Output: N
Algorithm: Transform N into a trigger function of the master function it is representing. A modified depth first search algorithm is employed

Order N so that $\underline{A}$ is on top
RecurseBDD_TransformTrigger(N .root, $\underline{U}$)
}

RecurseBDD_TransformTrigger(n, U)
{
if($n \rightarrow left$ points to a $u \in \underline{U}$)
then make $n \rightarrow left$ point to 0
else if($n \rightarrow left$ points to 1 or $n \rightarrow left$ points to 0)
then make $n \rightarrow left$ point to 1
else
RecurseBDD_TransformTrigger(n->left)

if($n \rightarrow right$ points to a $\in$ $u \in \underline{U}$)
then make $n \rightarrow right$ point to 0
else if($n \rightarrow right$ points to 1 or $n \rightarrow right$ points to 0)
then make $n \rightarrow right$ point to 1
else
RecurseBDD_TransformTrigger($n \rightarrow right$)

Enforce BDD rules
}

Figure 3.2. BDD Trigger Transformation Algorithm

Example: Suppose we had the function $f = \overline{a}\overline{b} + a\overline{b}cd + abcd$ and wished to have $\underline{A} = a, b, c$. This steps to accomplish this are shown below. Note that the notation $a_0 b_1$ refers to a path traversed in the BDD. This path starts at a, takes the left/zero path to b, and then takes the right/one path of the b node just reached.

- Step 1: Form a BDD, and reorder such that $\underline{A}$ forms the top variables.
- Step 2: Follow path $a_0 b_0$. Points to 1. No change.
- Step 3: Follow path $a_0 b_1$. Points to 0. Have b_1 point to 1. Mark b as visited. However, the BDD rules will force b to be removed and a_0 point to 1.
- Step 4: Follow path $a_1 b_0 c_0$. Points to 0. Have c_0 point to 1.
- Step 5: Follow path $a_1 b_0 c_1$. Points to d. Have c_1 point to 0. Mark c as visited.
- Step 6: Follow path $a_1 b_1 c_0$. Points to 0. Have c_0 point to 1.
- Step 7: Follow path $a_1 b_1 c_1$. Points to d. Have c_1 point to 0. mark c as visited. BDD rules will force the two c subtrees to merge as they are isomorphic. It is now obvious that b had to isomorphic children and therefore will be replaced by the subtree. No further changes are performed The trigger function of f is $T = \overline{a} + a\overline{c}$.

We are essentially removing cubes that do not only involve members in $\underline{A}$. Since this is a modified depth first search, this algorithm run in linear time with respect to replacements. It is known that BDDs have an exponential big-O space complexity [4], but often times this size is not reached.

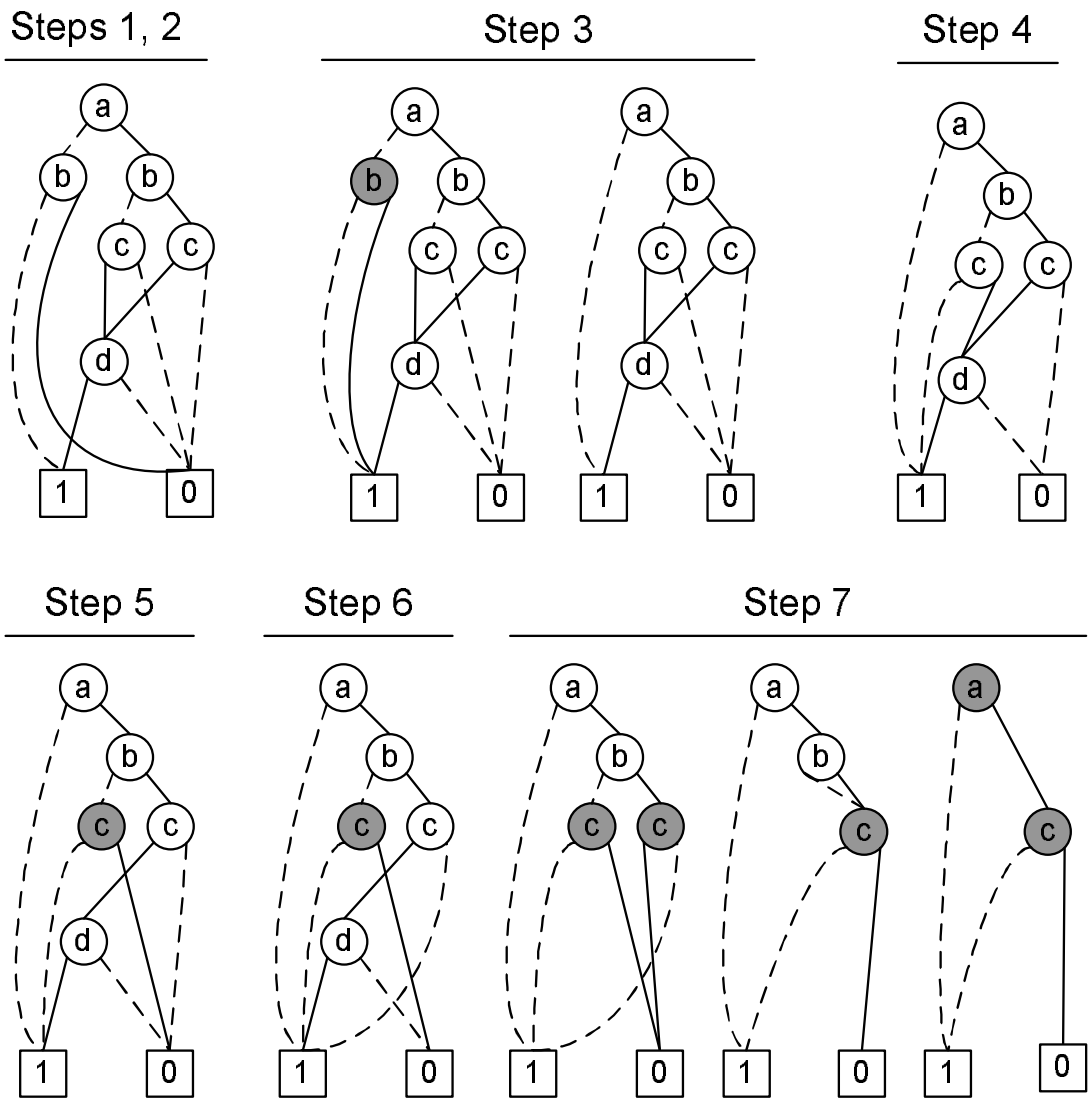


Figure 3.3. BDD Transformation Example

3.3.3. Multi-valued Logic Approach

In the above approaches, we assumed a particular $\underline{A}$. However, there are potentially many $\underline{A}$'s that also form trigger functions. These other trigger functions may cover EE opportunities that are not recognized by the originally chosen trigger function. Therefore, the use of more trigger functions will cover more early evaluation opportunities for a given master function.

A PL EE gate consists of the original master function PL cell augmented with another PL cell [36] (along with a small amount of control circuitry between the two). Figure 3.4 contains a diagram of the organization of a PL EE gate with a single trigger function.

In order to effectively utilize multiple trigger functions, the control mechanism between the master function and the multiple trigger functions must be augmented resulting in a new organization of a PL EE gate. A naïve implementation of such a gate would be to replicate trigger function cells for each candidate trigger function and then use a large fan-in OR gate to concentrate their outputs into a single signal causing the master function to early evaluate as shown in Figure 3.5. While this resulting circuit is functionally correct, this approach is too costly in terms of area since a considerable amount of circuitry in the trigger functions could potentially be shared.

It is desired to use a single super trigger function by forming a function that is the union of the on-sets of each individual trigger function allowing for term sharing to be exploited. If the outputs of the corresponding C-Muller elements are also ORed together, as shown in Figure 3.6, "false" EE can occur.

A "false" trigger can occur since the trigger function should evaluate only when all dependent variables are present. However, an individual C-Muller element may evaluate when only a subset of variables of the super trigger function are present.

To illustrate this problem, consider a master function $f(a, b, c, d)$ with individual trigger functions $t_1(a, b, c) = ab+bc$ and $t_2(a, b, d) = abd$. If the organization in Figure

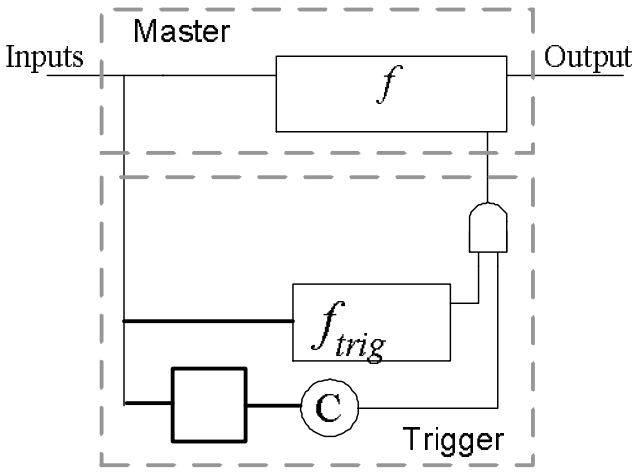


Figure 3.4. Single Trigger PL

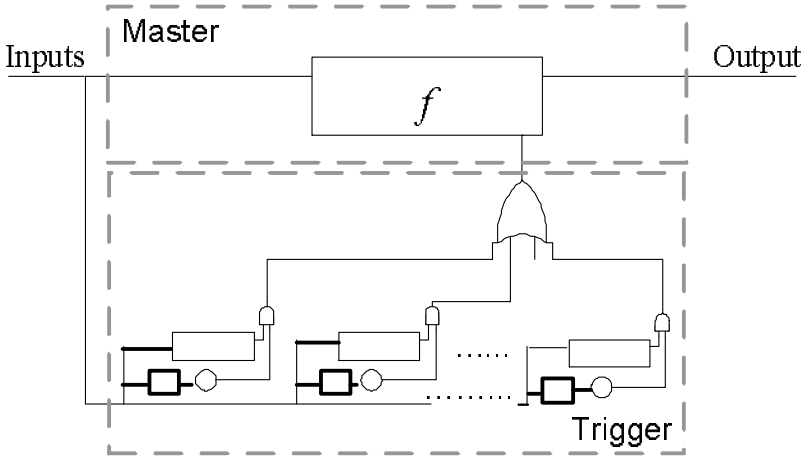


Figure 3.5. Naive Implementation of Multi-trigger EE PL Gate

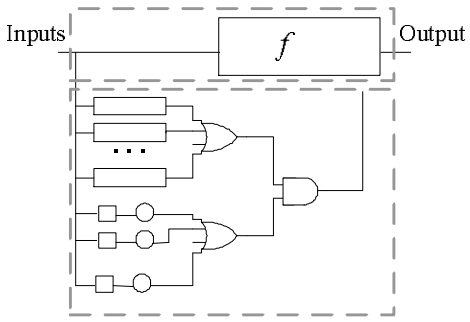


Figure 3.6. Incorrect Multi-trigger EE PL Gate

3.6 were used, the super trigger function is formed as $t_{sup} = t1 + t2 = ab + bc + abd$ and the two C-Muller elements (supporting t_1 and t_2) evaluate when variables $\{a, b, c\}$ are all present (corresponding to t_1) while the second C-Muller element will evaluate when variables $\{a, b, d\}$ are present.

Suppose that the signals corresponding to variables $\{a, b, d\}$ have arrived with values $\{1, 1, 0\}$ respectively and that variable c has not yet arrived. In this case, the C-Muller element corresponding to trigger t_2 will evaluate although the trigger function t_2 has value 0. However, t_{sup} has a value of 1 causing the master function f to incorrectly evaluate early although the signal corresponding to variable c has not arrived. In such a case, a false trigger occurs causing the master function to evaluate before it should have.

To solve this problem, a notion of "which" variables in the support of t_{sup} have arrived is needed. With a single trigger function, this notion is not needed since the single C-Muller element will evaluate only when all inputs corresponding to the trigger function are valid. Our approach is to formulate the super trigger function such that the individual C-Muller elements are combined into a single *binary valued-function* that *depends on ternary-valued variables resulting* in t_{mvl}. The organization of this form of the EE PL gate is then depicted as shown in Figure 3.7.

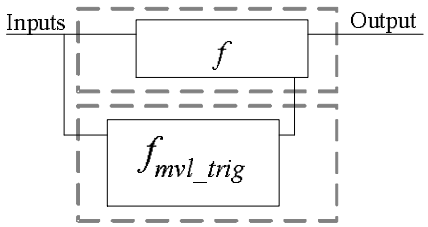


Figure 3.7. PL EE Gate with MVL-based Multi-trigger Function Support

The notion of signal availability is incorporated into the PL EE Gate depicted in Figure 3.7 by using the dependent ternary-valued variable encoding shown in Table 3.1.

The *multi-valued logic* (MVL) super trigger function t_{mvl} is formed as the disjunction of terms that represent each individual trigger function and the availability of their respective inputs. Each such term is the conjunction of the individual trigger functions and a corresponding expression indicating input signal availability. The expressions representing input availability effectively take the place of the C-Muller elements and are represented as MVL functions C_i.

Using the example shown above that precludes the organization as shown in Figure 3.7, the individual trigger functions and their corresponding C_i functions become:

$$t_1(a,b,c) = a^{\{1\}}b^{\{1\}} + b^{\{1\}}c^{\{1\}}$$

$$C_1(a,b,c) = \overline{a^{\{2\}}}\ \overline{b^{\{2\}}}\ \overline{c^{\{2\}}} = a^{\{0,1\}}b^{\{0,1\}}c^{\{0,1\}}$$

$$t_2(a,b,d) = a^{\{1\}}b^{\{1\}}d^{\{1\}}$$

$$C_2(a,b,d) = \overline{a^{\{2\}}}\ \overline{b^{\{2\}}}\ \overline{d^{\{2\}}} = a^{\{0,1\}}b^{\{0,1\}}d^{\{0,1\}}$$

Using these formulations, the super trigger function t_{mvl} is expressed as:

$$t_{mvl} = t_1C_1 + t_2C_2$$

$$t_{mvl} = a^{\{1\}}b^{\{1\}}c^{\{0,1\}} + a^{\{0,1\}}b^{\{1\}}c^{\{1\}} + a^{\{1\}}b^{\{1\}}d^{\{1\}}$$

It is now easy to see that the cube $a^{\{1\}}b^{\{1\}}c^{\{2\}}d^{\{0\}}$ corresponding to the scenario described previously will result in t_{mvl} evaluating to 0.

Table 3.1. Ternary Encoded PL Signals

Logic Value	Interpretation
0	Signal arrived with binary value 0
1	Signal arrived with binary value 1
2	Signal has not yet arrived

The actual implementation could be accomplished using MVL circuitry, or, by mapping back to a binary circuit where dual-rail lines are used for each variable. Although the binary-mapped version may appear to double the amount of required wiring, we note that signals are already present in dual-rail form in order to support the LEDR encoding used in PL circuits. This increase in wiring for the inputs of the super trigger portion of the EE PL gate is less overhead overall than would be present in the naïve approach of ORing all individual single trigger function blocks as shown in Figure 3.5.

Chapter 4

EARLY EVALUATION - IMPLEMENTATION

There is a need for EE identification to be done automatically, which should require minimal effort by the designer to determine trigger functions for insertion in a PL design. As EE is a local transformation on combinational portions of a netlist, we may perform EE identification on either a clocked or PL netlist. We have chosen to perform such identification on a clocked netlist as it makes the PL mapper implementation less cumbersome. The implementations to follow are the tools needed to find such candidates in an automatic fashion.

4.1. Fine-Grained Based Implementation

The fine-grained PL tool suite creates circuitry consisting of 4×1 $look-up\ tables$ (LUT4s) and $1-bit\ flip\ flops$ (DFFs). In this framework, an n-bit logic function will be decomposed into various 4-input functions. As such, the logic functions that an automatic, fine-grained EE inserter must be able to handle are 4-input functions. Since available sets of size zero or four would be trivial, non-trivial trigger functions of a given LUT4 are either three, two, or one variables in size. So, the total number of non trivial trigger functions is $\begin{pmatrix} 4 \\ 3 \end{pmatrix} + \begin{pmatrix} 4 \\ 2 \end{pmatrix} + \begin{pmatrix} 4 \\ 1 \end{pmatrix} = 14$. One can easily compute all 14 trigger functions and then compare the merits of each to determine the best choice. In the results for a fined-grained EE inserter, the trigger function is determined using the truth table approach. As described previously, this approach is asymptotically exponential in time and space, however, is adequate for functions of size 4.

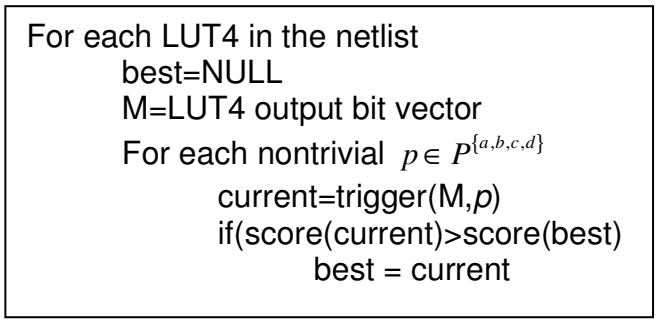


Figure 4.1. Pseudo Code of Fine-Grained Trigger Identification Function

To determine master/trigger functions pairs, the netlist of the clocked design must be processed. This netlist is composed of DFFs and LUT4s, where the components of interest are the LUT4s. Each LUT4 is specified by various identifier information, netlist connections, and logic behavior. The logic behavior is specified by an output bit vector. This bit vector is the main input into the EE identifier code `trigger()`, which returns a trigger based on the output vector and a variable set. As a LUT4 may have various trigger functions to choose from, a score function is utilized to determine which trigger function is the most appropriate.

4.1.1. Pseudo Code

Suppose for an arbitrary LUT4, the inputs are a, b, c, d. The basic algorithm for determining master/trigger pairs is shown in Figure 4.1. The `trigger()` function determines what a trigger function with respect to `p` is, if it exists at all. The `trigger()` function attempts to find a trigger function by looking at the logic input combinations and corresponding output behavior. It will group logic input combinations that have similar assignments after the $\underline{U}$ variables are treated as don't cares. If the output behavior i of the set of similar assignments are the same that assignment is given a 1 in the T[i], else it is assigned a 0.

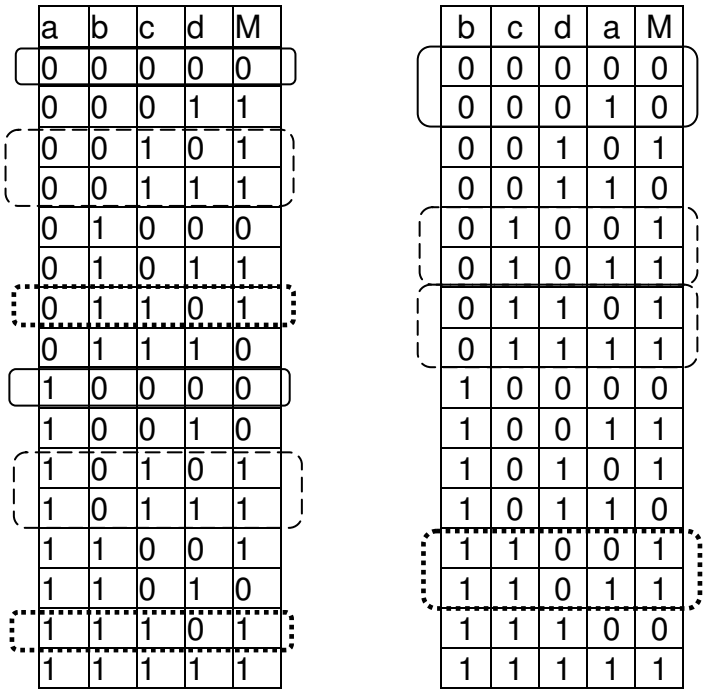
```
trigger(M,p)
{
        M' = Reorder(M,p)
        For( i = 0 ,  j = 0  ; i < |p|  ; i = i + 2^(|M|-|p|) ,  j = j + 1 )
                if(M'[i...i + 2^(|M|-|p|)] == [1...1] or M'[i...i + 2^(|M|-|p|)] == [0...0])
                        T[ j ] = 1
                else
                        T[ j ] = 0
        return T
}
```

Figure 4.2. Psedocode of trigger() function

To aid in searching for input combinations with the described similar assignments, the inputs are ordered with the $\underline{A}$ variables as the higher bits, and $\underline{U}$ the lower bits, as shown in Figure 4.3. This ensures that the similar input combinations are grouped next to each other. Additionally, each group will be of size $2^{|M|-|p|}$. Since there are only 16 input combinations, we do not need to look very far to determine a trigger function.

4.1.2. Score Function

In determining the goodness of one trigger function as compared to another, provided by the `score()` function, two criteria are used: coverage and input arrival times. Coverage refers to the number of minterms the trigger function covers in both the on- and off-set of the master function. The higher the coverage, the more likely an assignment to the trigger variables will result in an early evaluation. Input arrival times indicate whether a particular input should be part of the input set of the trigger function. Namely, one would rather have fast arriving inputs than slow arriving inputs. However, a fast input may not offer much coverage. A score function is used to try and balance the constraints. If two scores happen to be the same one may em-



a	b	c	d	M
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

b	c	d	a	M
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Figure 4.3. Example of Variable Reordering

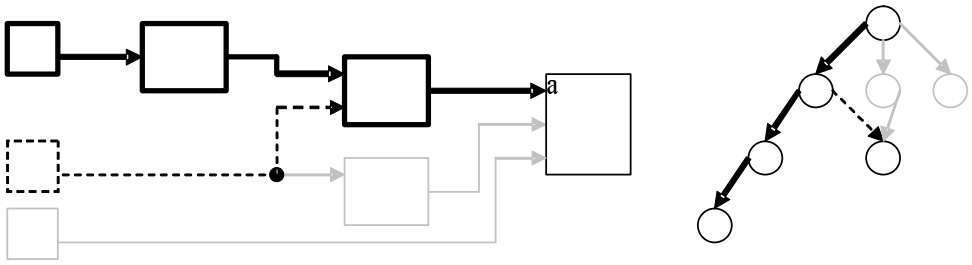


Figure 4.4. Example of Arrival Time Determination

ploy a greedy approach or use further heuristics to determine the better candidate. A greedy approach was used in the implementation presented in this work.

Arrival times in a PL system are dependent on the topology of the associated graph as well as the initial token marking. However, the netlist file that is processed is of the clocked system. So a heuristic method for arrival time estimation is used. The basic computation used in determining arrival times is merely the summation of gate delays between the primary circuit input to the farthest "input" DFF. Namely, one does a breadth-first traversal from the primary circuit input going towards its' driver gate, and continues doing this using DFFs or external inputs as a stopping condition. The longest path found is the path used to determine the inputs' arrival time. Figure 4.4 illustrates how the arrival time for input a is determined.

We find such a time for all function inputs. We denote the max of all the input times as M_{max_time}. For a potential $\underline{A}$, we denote the max input times of the $\underline{A}$ as T_{max_time}. It is desirable to have a T_{max_time} that is smaller than M_{max_time}, and the term $M_{max\_time} \cdot T^{-1}_{max\_time}$ embodies this notion. Additionally, we wish to have as high a coverage as possible. A score function that displays these characteristics is $score = Coverage \cdot M_{max\_time} \cdot T^{-1}_{max\_time}$.

4.1.3. Experimental Results

Several benchmark circuits were synthesized both with and without the use of the EE algorithm. The benchmarks used were the *International Test Conference 1999* (ITC99) suite [8]. These are available in RTL level VHDL format and were synthesized using the Synopsys Design Compiler tool. The resulting EDIF netlist was then mapped to PL technology using the tool described in [29] and then post-processed for the inclusion of EE circuitry. The delay value for each component in the PL netlist is 7 ns. For the 15 benchmarks presented in Table 4.1, an average speedup of over 13% was achieved with an average increase in the amount of circuitry for the EE gates resulting in 33%. In these results, EE circuitry was added to all PL gates where a speedup was possible. It is also possible to reduce the increase in area by requiring a candidate trigger function to have a score value that exceeds some threshold. Thresholding the score function allows for a tradeoff in area versus delay of a PL circuit. Table 4.1 contains columns representing the description of the benchmark circuit, the number of PL gates required without EE, the number of EE gates when the circuit is synthesized using EE, the average delay with no EE, the average delay with EE, the difference between the average delay with and without EE, the percent of area increase in terms of additional gates when the EE algorithm is applied and the percent decrease of delay when EE is used in the synthesis of the benchmark circuits. Note that the area values do not include increased area due to LEDR signaling. The delay results are based upon the average statistics of 100 simulations where the input vectors were randomly generated. For each PL circuit, we determined the average delay time between the presence of a stable input vector and a stable output word. In a PL circuit, new values cannot be presented to the inputs until a stable output is generated for the current input values. Furthermore as is discussed in [26], delays are statistically distributed based on the value of an input vector and are not constant in PL circuits. To determine the delay values between the PL circuits with and without EE, we computed the average of the difference between

Table 4.1. Fine-Grained Experimental Results

Name	PL Gates (no EE)	EE Gates	Avg. Delay w/o EE (ns)	Avg. Delay w/ EE (ns)	Delay Diff (ns)	%Area Increase	%Delay Decrease
B01	25	9	49	43	6	36.00	12.24
B02	4	0	18	18	0	0.00	0.00
B03	78	25	49	50	-1	32.05	-2.04
B04	274	102	84	85	-1	37.23	-1.19
B05	322	136	98	88	10	42.24	10.20
B06	10	1	26	27	-1	10.00	-3.85
B07	240	95	87	67	20	39.58	22.99
B08	82	24	66	52	14	29.27	21.21
B09	74	23	46	45	1	31.08	2.17
B10	126	49	63	59	4	38.89	6.35
B11	275	112	132	93	39	40.73	29.55
B12	635	263	80	73	7	41.42	8.75
B13	141	44	56	51	5	31.21	8.93
B14	3360	1565	332	207	125	46.58	37.65
B15	5648	2611	336	184	152	46.23	45.24

the cycles for both circuits. Mentor Graphic's `qhsim` was used to simulate the PL VHDL we generated for each test bench. Because a master/trigger pair of PL gates requires the use of an additional Muller-C element, some benchmarks suffered a slight degradation in overall delay values when the EE algorithm was applied. Overall, the EE algorithm resulted in a speedup for most of the benchmarks. Not surprisingly, those benchmarks with significant amounts of arithmetic circuitry tended to take more advantage of the EE algorithm since arithmetic circuits tend to be composed of addition circuits where EE techniques are known to perform well.

```
triggerBDD( f , A )
{
        For each  A ∈ A
                FBDD = ROBDD( f ,  A )
                best = NULL
                B = BDD_ConstructTrigger(FBDD)
                if(score(FBDD)>score(best))
                        best = FBDD
}
```

Figure 4.5. Pseudo code for Large Function Trigger Identification

4.2. Trigger Functions via BDD Implementation

As discussed in Chapter 3, the BDD algorithm is used to handle master functions of large size. Since the procedure for determining a trigger function works on a functions representation, a compact and efficient means of interacting with the representation is essential. The *Colorado University Decision Diagram* (`CUDD`) [32] package is a widely known BDD library that has both characteristics and is used in this implementation.

4.2.1. BDD Method for Trigger Functions

The implementation is much like the truth table implementation except that the method of trigger detection is replaced by the BDD approach. Pseudo-code for an implementation is shown in Figure 4.5.

Note that **A** is a set of interesting $\underline{A}$. In the truth table approach, trigger functions for all non-trivial $\underline{A}$s is done. However, as n grows large, this becomes impractical due to the large number of non-trivial $\underline{A}$'s. Therefore, a heuristic for determining $\underline{A}$s to look at must be formulated. Using timing information for the inputs may suggest which variables to place in $\underline{A}$.

Because the trigger function must depend upon a proper subset of dependent variables of the master function, all cubes in a candidate trigger function correspond directly to the 1-paths and 0-paths in the master function BDD. Such paths are easily extracted from the master function BDD through a single traversal. In order to incorporate the timing constraint, we reorder the variables in the BDD such that those variables corresponding to minimum arrival time signals are first in the ordering. This approach effectively translates the process of extracting trigger functions to variable reordering of a BDD although we are not using the typical constraint of BDD minimization to perform reordering. In the event that the BDD representing the master function exceeds some preset size limitation, the master function can be partitioned into two new master functions and the process repeated.

This algorithm is similar to the transformation algorithm presented in the previous chapter however, the master function is not modified, but a separate trigger BDD is constructed. This is convenient as the master function may be reused to find other possible trigger functions. A relationship is created between the trigger nodes n' and the master nodes n that correspond to that portion of the trigger. This correspondence takes the place of a visit flag used in the transformation algorithm.

4.2.2. Experimental Results

The BDD-based trigger function extraction experiment was carried out by constructing a BDD using the CUDD software and by using the MCNC benchmark circuits in `.pla` format. No variable reordering is applied to the BDDs as it was assumed that those variables higher in the assumed order correspond to earlier arriving inputs. Each output was considered to represent a single-output master function and trigger functions were extracted that depend upon the first $\lfloor j/2 \rfloor$ variables in the BDD ordering where j represents the total number of variables in the BDD. The choice of $\lfloor j/2 \rfloor$ variables in the trigger functions support set is arbitrary.

```
BDD_ConstructTrigger( N , A , U )
{
        INPUT:      BDD N , A , U
        OUTPUT:     BDD N'
        Algorithm:  Perform a modified depth-first search on N to
                    construct a trigger function N'

        Order N with the A at the top
        Let ∀n ∈ N , n.n' = NULL     // n' is the node in trigger that
                                     // correspond to n
        RETURN N' = Recurse_FindTrigger( N .root, U )
}

RecurseBDD_ConstructTrigger( n , U )
{
        IF( n ≡ 1-node or n ≡ 0-node)
                THEN RETURN 1-node
        IF( n.n' )
                THEN RETURN n.n'
        n.n' = copy( n )
        IF( n ∈ U )
                THEN RETURN n.n' = 0-node
        L = BDD_FindTrigger( n.0 , U )
        R = BDD_FindTrigger( n.1 , U )
        IF( L ≡ R )
                THEN RETURN n.n' = L
        n.n'.0 = L
        n.n'.1 = R
        RETURN n.n'
}
```

Figure 4.6. Pseudo Code of BDD_ConstructTrigger()

Table 4.2. BDD Implementation Experimental Results

Name	in/out	k	Coverage
addm4	9/8	5	0.781
majority	5/1	3	0.125
cordic	23/2	10	0.758
sao2	10/4	6	0.828
squar5	5/8	3	0.875
dist	8/5	4	0.625

Table 4.2 contains the experimental results. The computer runtime required to compute the trigger functions in Table 4.2 all required less than 1 ms. This is not surprising since the algorithm is merely a modified depth-first traversal, which is known to be $O(N)$, with N being the number of vertices in the BDD. Column one contains the name of the benchmark circuit. Column two shows the total number of inputs/outputs. Columns three and four pertain to the first circuit output in each netlist. The third column contains the number of dependent variables in support of the trigger function (k) (maximum possible value of k is $\lfloor j/2 \rfloor$ and the minimum is 0). The fourth column contains the percentage of minterms in the on-set of the trigger function divided by the number of all those possible (i.e. on-set minterms divided by 2^j). This latter value is a measure of the coverage value that the trigger function provides.

4.3. MVL Trigger Implementation

In the above approaches only one trigger function of potentially many is selected. An MVL trigger may utilize several trigger functions to increase coverage. The process for creating MVL triggers is divided into two stages based on the tools used.

1. Find individual trigger function and create an MVL trigger.
    - *Custom tool using* `CUDD` (CT)
2. Minimize the MVL trigger.
    - `MVSIS` [11]

Since `CUDD` does not operate on MVL functions, another means must be used to transform the Boolean trigger functions into MVL trigger functions. The OR operation then needs to be applied to the MVL triggers to form one super MVL trigger. Minimization is then applied to decrease the potential cost of an implemented MVL super trigger.

A description of the CT tool is given below:

1. Find triggers for a particular master function
2. Collected the best triggers
3. Turn the triggers into MVL versions
4. Create a super trigger, which is the OR of MVL triggers

The first step is to collect various trigger functions from the master function. This may be done by repeated application of the BDD approach using various variable subsets. Aside from the possible heuristics used in the BDD approach, different constraints may be added in the MVL approach. For instance, if a trigger is found to be good for a particular variable subset, one may not want other trigger functions that depend mainly on those same variables. Using different triggers with different variable subset may create MVL triggers with higher coverage. Additionally, certain inputs may be slow or fast depending on the state of the system. In these cases, it may be desirable to have triggers that account for this difference. Another method may be to merely select the triggers with the highest scores. However, these triggers may overlap in coverage if they use a lot of the same variables in the support set.

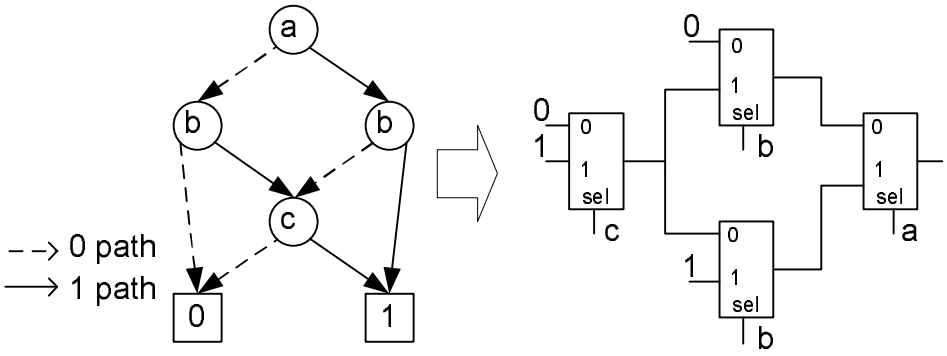


Figure 4.7. Transformations of a BDD to a Multiplexer Network

Considering this, it is best to find trigger functions with high scores, but that also have a large amount of disjointness in coverage.

After BDD triggers are found, they are output in `blif` netlist format [37] by `CUDD`; however, this format is not adequate to generate MVL trigger functions. The `blifmv` format [15] is capable of specifying MVL circuitry, but is not a format that `CUDD` is able to produce. Therefore, a parser is used to turn the functions represented in `blif` into MVL triggers formatted in `blifmv`. We then generate another MVL function that represents the OR of all the individual MVL triggers.

`CUDD` uses a simple transformation of a BDD to a network of binary multiplexers when it outputs a function in `blif` format, where each node is a multiplexer and the associated variable is a select line. To turn this multiplexer network into an MVL network, we must consider that each variable now has three possible values: 0, 1, or 2. With respect to the tree-structure of the BDD, we must give each node an additional 2-branch that points to the 0 terminal node. This structure is no longer a BDD; it is a ternary multi-decision diagram (MDD). Note, the implementation never processes an MDD, it is only presented to show the correlation between a binary multiplexer network and a ternary multiplexer network.

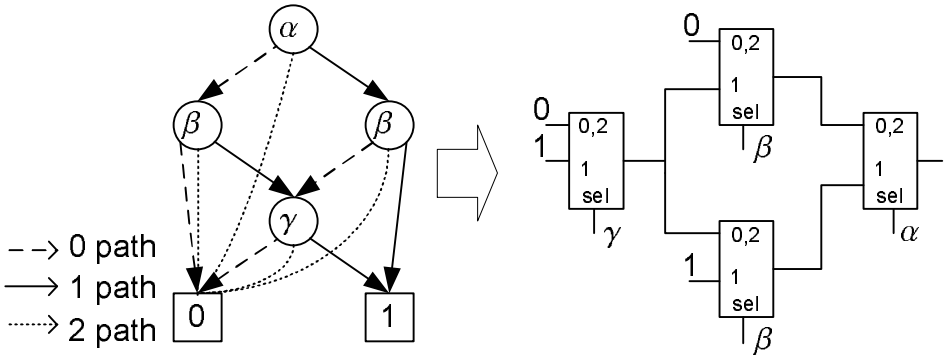


Figure 4.8. Transformation of a Ternary MDD to a Ternary Multiplexer Network

The procedure takes a netlist of multiplexers and transforms them into multi-valued multiplexers. Additionally, in order for the proposed MVL structure to work, all variables must have arrived for a particular single trigger. This constraint is added to a MVL trigger function circuit. To create the super trigger, we take all MVL single triggers and OR them together to form the super MVL trigger.

One may observe that the super trigger structure so far described is similar in appearance to the proposed naive approach. However, since the structure is in MVL form, it is possible that logic minimization may occur. `MVSIS` is used to minimize the MVL trigger function.

4.3.1. MVL Experimental Results

In this set of experimental results, super trigger functions are found and compared to single trigger functions using the BDD method. The experiments were carried out by using the MCNC benchmarks to represent master functions with the first output being used to represent the master function output. Multiple trigger functions were generated using random subsets of variables. These trigger functions were then used to form an MVL trigger function. The results of these MVL trigger functions were then compared to the best single trigger function within the set of possible trigger

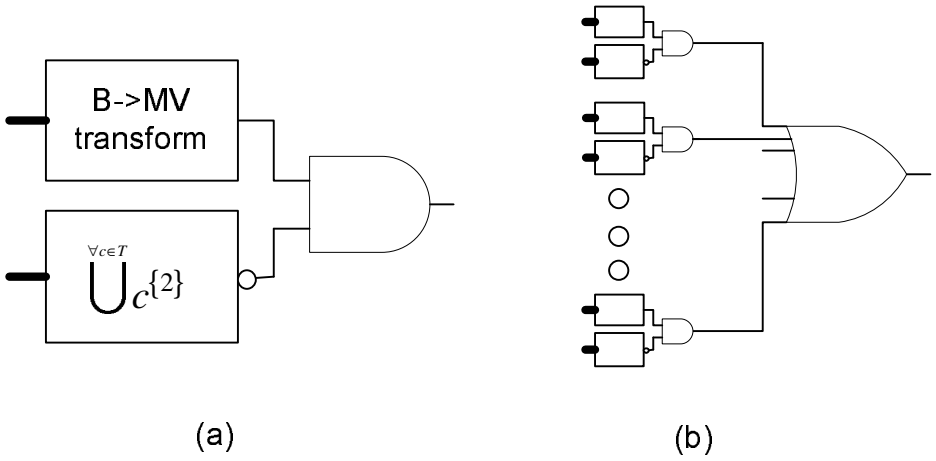


Figure 4.9. (a) Single MVL Trigger and (B) Super MVL Trigger

functions. Table 4.3 contains these experimental results. The first column contains the name of the benchmark (master) function, the second contains the inputs/cubes of the master function, the third contains the inputs/cubes of the single trigger function found using the BDD method the fourth contains the coverage achieved by the single trigger function, the fifth contains the inputs/cubes of the multi-trigger function, and the final (sixth) column contains the coverage provided by the multi-trigger function.

These results are given after `espresso` [2] was used to minimize all functions after `MVSIS` [11] was used to map the t_{mvl} function back to binary form.

4.4. Summary

An improvement in the formation of EE PL gates is described where a BDD-based method allows for the extraction of trigger functions from large master functions is demonstrated as effective as compared to the exhaustive truth table method. The coverage provided by a trigger function was also enhanced by utilizing multiple trigger functions for the same master function and then using MVL methods to combine them into a super trigger function.

Table 4.3. MVL EE Trigger Experimental Results

Name	Master	Single EE		MVL EE	
5xp1	7/7	3/3	0.75	6/3	0.75
addm4	9/9	2/2	0.75	14/6	0.83
majority	5/5	3/2	0.63	6/2	0.83
cordic	23/143	8/10	0.63	18/33	0.63
sao2	10/10	6/7	0.55	14/14	0.59
squar5	5/2	3/3	0.81	8/4	0.84
dist	8/12	5/3	0.88	12/5	0.94

Chapter 5

SLACK MATCHING BUFFERING

The work presented here describes a performance enhancement technique for PL, the automatic insertion of *slack matching buffers* [9]. Slack matching buffers perform no logical operation on the data signals, rather they serve as intermediate signal storage locations in a circuit thus allowing other portions of the circuit to continue to process available data signals and to decrease the number of localized "stall" situations. The idea of using slack matching buffers for asynchronous circuit performance enhancement is not new and has been described in past work [18]. In terms of PL, slack matching buffer insertion was identified as a viable performance enhancement method in [26]; however, an automated process for buffer insertion was not devised. In subsequent PL design efforts, slack matching buffer insertion was accomplished manually via ad hoc methods based on the designer's detailed knowledge of the circuit functionality.

A technique for automatic insertion of slack matching buffers was devised through the use of a custom PL circuit timing simulator. This simulator models the performance characteristics of a PL circuit only and does not provide any functional information. Active portions of the circuit at any instant in time are represented as the flow of "tokens" representing the data, request, and acknowledge signals. This approach allows the simulator to have significantly faster run times as compared to a traditional functional simulator.

Based on the analysis of PL circuits using the simulator, an automated method for slack matching buffer insertion is formulated that allows buffers to be inserted based solely on the topology of the PL circuit. Because the automatic buffer insertion

technique utilizes the token flow simulations, a crucial aspect of this work is the efficiency of the custom simulator tool.

Note, as opposed to the generic PL design flow shown in Chapter 2, slack matching buffer identification is not done on the clocked netlist. We must first have a PL system, and then simulate to find buffer insertion locations. Ideally, identification of slack matching buffers should occur before the mapping stage as we can directly convert the clocked netlist to a PL netlist and reduce the number of checks for safety in the induced marked graph.

5.1. Slack Matching Buffering

As defined in [18], the *slack* of an asynchronous pipeline is the number of tokens that may be placed in the pipeline before it stalls. Additionally, the modification of slack properties in asynchronous pipelines may change the performance of the circuitry; this type of modification is a well-known optimization technique referred to as *slack matching*. In terms of PL, we wish to be able to adjust the property of slack in order to improve performance by maximizing available parallelism between various paths. The goal of this work is to characterize where slack matching buffers can improve performance and then to automatically insert them into the netlist.

5.2. PL Token Flow Simulator

In order to better understand the behavior of token movement with respect to the topology of a marked graph, a simulator was developed. The simulator allows for direct manipulation of token placement as well as nodes. This allows for the setup of experiments to characterize the behavior of the placement of token buffers with a given topology. Additionally, the simulator is able to accept various, custom netlist formats of real designs and simulate them.

As there are numerous Petri net based simulators and asynchronous circuit design tools [25], the contribution of this simulator is the ability to measure latency and

throughput of a PL graph. To do this, we have to add token sources and sinks to the marked graph that represent overall token producers and consumers of the entire circuit. Whenever a source fires, it provides a tag along with the token it places on the output. The tag contains information such as firing time and source id. This tag will propagate through the netlist and join with other tags from other sources, if encountered, to form larger tags. Once the tag reaches a sink, information such as latency may be computed using the tag.

The simulator uses an event-driven approach. A queue is maintained that keeps track of nodes that are ready to fire. A node is ready to fire when all its inputs have a token. Before the simulation starts, the queue is populated with nodes that are ready to fire based on the initial token mapping. The nodes of a marked graph may be either transitions or places as described in standard Petri net theory. These correspond to gates and signals of a physical circuit.

Due to the nature of physical circuits, gates and signals may both have non-zero delay. We allow the possibility for varying delay of these components in the simulator. To allow for this we have a notion of place firings as well. When a transition fires, the transition inserts its output places into a priority queue called `delay_fire`. When a place fires, we check if this firing would allow the connected transition to fire. If so, we place the transition into the `delay_fire` queue. After a node has fired, it is removed from the queue.

Based on a nodes' delay value, we assign the node to fire at `current_time + delay`. The simulator checks the `delay_queue` at each time instant to see if a delay firing needs to occur. If so, the node is fired and removed from the queue.

The approach for inserting slack matching buffers is reliant on throughput and latency information provided by the simulator. To get this information, several simulation cycles must be computed. However, this computation is not too costly since the data quickly converges to stable values that may be used for slack matching buffer insertion.

To justify the use of a simulator in a synthesis tool, the simulator must be very fast. Let us assume a marked graph with t transitions and p places. An upper bound for nodes ready to fire is $t+p$. So the simulator must perform $t+p$ firing operations. A firing operation consists of checking whether to add the node driven needs to go into the `delay_fire` queue. We know that a place may only input into one transition, so that is one check. If we suppose the transitions form a t-complete graph, we need $t-1$ checks for each transition. This means $t(t-1) = t^2 - t$ checks. In total, there are $t^2 - t + p$ checks, giving a $O(t^2)$ complexity for firing checks per iteration. If we assume more realistic graphs, such as transitions with a maximum of a fanout of 4 we would get $O(4t+p)$.

For realistic circuits, each simulation cycle has linear time complexity. However we only need to run the simulator to get latency and throughput times, which converge fairly quickly. Empirical results suggest as few as 10 iterations may provide good results. Additionally, 10 iterations frequently take less than a second, even for large graphs.

5.3. Results

A gate's feedback wait time at cycle i, ω_i, is the difference between the time that all data tokens are available at the inputs of a PL gate and the time when all feedbacks are available. If all feedbacks are available before the data, then we say $\omega_i = 0$. During a simulation a gate may fire n times. We denote a gate's average feedback wait time as $\Sigma\omega_i/n = \delta$. The rule to add token buffers depends on a gates average feedback wait time.

A branch and bound procedure used to determine slack matching buffer placement is the following:

1. Run the simulator on a topology to get δ for each gate
2. Collect the gates with high δ, call it set C

3. Determine a candidate for buffering from C

4. Add slack matching buffer and ensure the resulting topology is a marked graph

5. Re-run the simulator

   (a) If the buffer does not improve performance, remove the buffer

   (b) If the buffer improves performance, keep the buffer

6. Repeat steps until some threshold is met

In step 3, we need rules to determine a candidate from the set C. The priority for selecting gates is based on:

- Feedback latency
- Fan-out

Feedback latency, $fl_{i,j}$, is the time for the j^{th} feedback to become available after all data inputs have arrived on a given gate at cycle i. If the j^{th} feedback arrives before all data inputs then $fl_{i,j} = 0$. This means that $\omega_i = max(fl_{i,j})$ for a given gate. Fanout is a good indicator of slack matching buffer placement since a signal that fans out will result in more feedbacks needing to return to the gate. This increases the odds that the gate will have to wait on a feedback.

For the gates that have a high waiting time on feedbacks, a method for determining if a slack-matching buffer would be beneficial is the following:

1. Find the cycle that shares the feedback and a data output of the gate

2. If the gate that the data output is driving is waiting for other data, then place a slack-matching buffer on the data output. If the output fans out, place the buffer on the path between the output and the next gate on the cycle

3. Re-run the simulator to make sure throughput and latency have improved

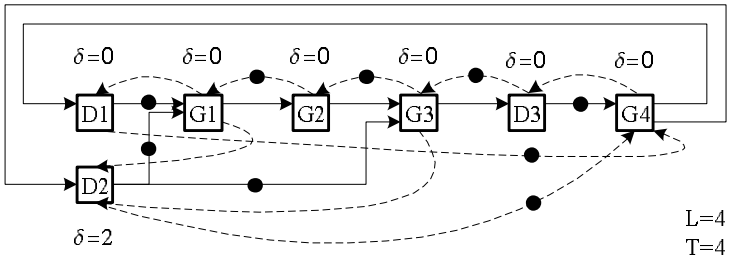


Figure 5.1. Example Topology with $L = 4$ and $T = 4$

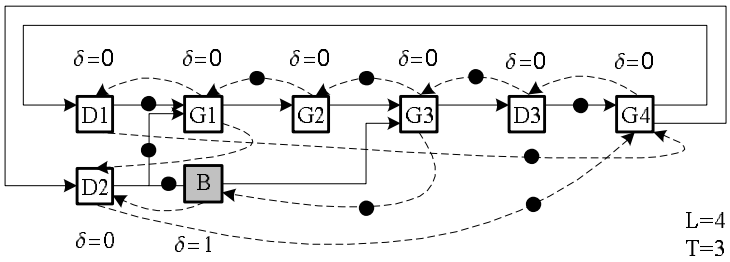


Figure 5.2. Example Topology with Slack Matching Buffer

We will use the following examples to denote how these rules are applied. Let us examine Figure 5.1 with latency $L = 4$ and throughput $T = 4$. Notice that $D2$ has a wait time of $\delta = 2$. In addition it is a fan-out point of two pipelines. However, $D2$ cannot evaluate until the acknowledge signal is received from $G3$ and is thus stalled. This implies that the slack-matching buffer should be placed between $D2$ and $G3$. If we add a slack-matching buffer at the output of $D2$, as shown in Figure 5.2, the delay characteristics improve. One may notice that B is now waiting; however, insertion of another buffer adds no performance improvement. Another example is shown in Figure 5.3.

The latency is $L = 8$ and average throughput is $T = 2.5$ for the circuit represented in Figure 5.3. If we apply our rules, candidates for application of slack matching

buffers at the outputs are $G2$, $G5$, and GD. It turns out that token buffers on $G2$ and $G5$ help performance, while a token buffer on GD does not. Figure 5.4 contains the topology of the circuit depicted in Figure 5.3 with the slack matching buffers included.

Topologies that do not benefit from slack matching buffer insertion are those that are purely serial in nature, or simple pipeline structures. An example of such a topology is shown in Figure 5.5. No matter where a buffer is placed, the throughput and latency do not improve. This is because there is no parallelism to take advantage of.

Table 5.1 provides results of the slack buffer insertion algorithm. Six benchmark circuits topologies were chosen for these experiments. The circuits were initially mapped into PL circuits without any performance enhancements and were later modified to contain slack matching buffers using the technique described above. The table contains the name of the benchmark circuit, and, in columns two and three, the initial throughput T and latency L. After application of the buffer insertion technique, columns three, four, and five contain the number of buffers inserted and the new throughput and latency values. The overall percentage of throughput improvement computed is given in the last column of Table 5.1. Over the set of benchmarks, an average improvement in throughput of 21% was achieved.

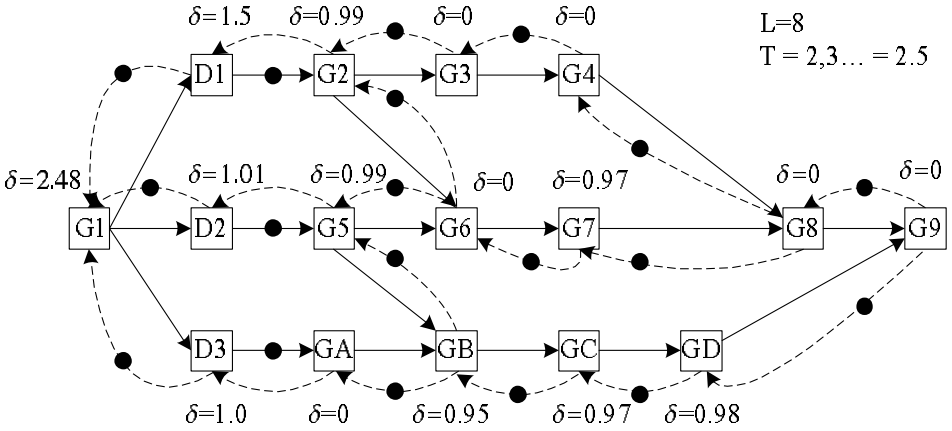


Figure 5.3. Second Example with $L = 8$ and $T = 2.5$

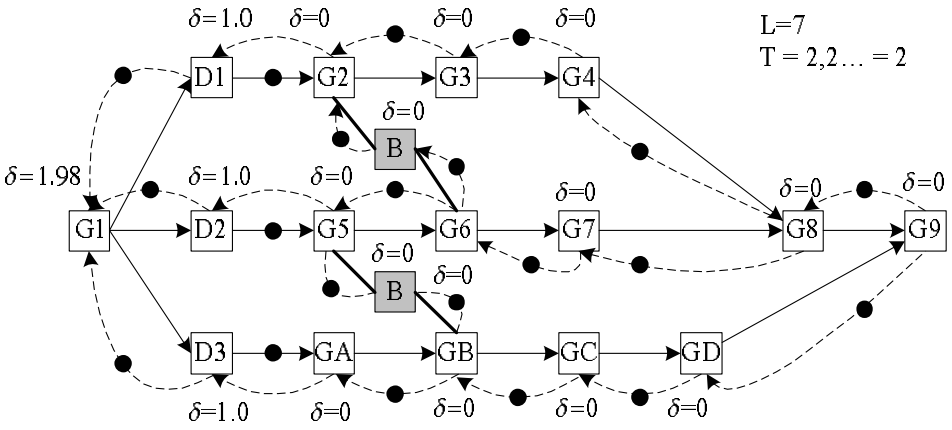


Figure 5.4. Second Example with Slack Matching Buffer

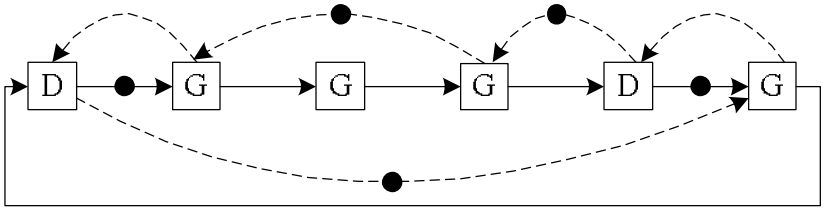


Figure 5.5. Single Pipeline Topology

Table 5.1. Slack Matching Buffer Experimental Results

Circuit	Original Latency L	Original Throughput T	Buffers added	New Latency L'	New Throughput T'	%Increased Throughput
4pipe	4	3.5	6	4	2	43
Tb3	10	2.33	2	10	2	14
T336	6	2.67	2	7	2	25
Tb	4	4	1	4	3	25
C17	4.5	2.5	1	4	2	20
S27	11	6	1	13	5	17
Average						21

Chapter 6

PLFIRE: PHASED LOGIC VISUALIZATION TOOL

With regards to PL optimizations, as different optimization techniques are implemented for PL, it becomes increasingly difficult to gauge the effectiveness of a technique without the use of some visualization tool. Current optimization techniques such as early evaluation and slack matching change the flow of data tokens with a PL circuit. Being able to visualize token flow allows a researcher to better understand how these techniques affect a circuit on a global level.

6.1. Visualization Tool

As the size of circuits that are being investigated in PL research grows, mentally keeping track of a designs PL structure and behavior becomes increasingly difficult. The purpose of the visualization tool, known as `PLFire` [10], is to help a designer visualize the behavior of a PL circuit. Figure 6.1 is a screen shot of the program.

Moreover, the use of `PLFire` aids in the development of optimization techniques and tools. As different optimization techniques are implemented for PL, a way of determining whether this optimization truly has an impact on the overall circuit must be achieved. Current techniques center on the ability to maximize throughput of token flow and performing global and local transformations of the netlist. Being able to see these optimizations gives the researcher some understanding of their overall affects. Furthermore, this tool will allow us to investigate the interoperability of these techniques.

In order to use the tool, two previously generated files are needed: a `viznet` file and a `viztiming` file. The `viznet` file contains the netlist information of the PL

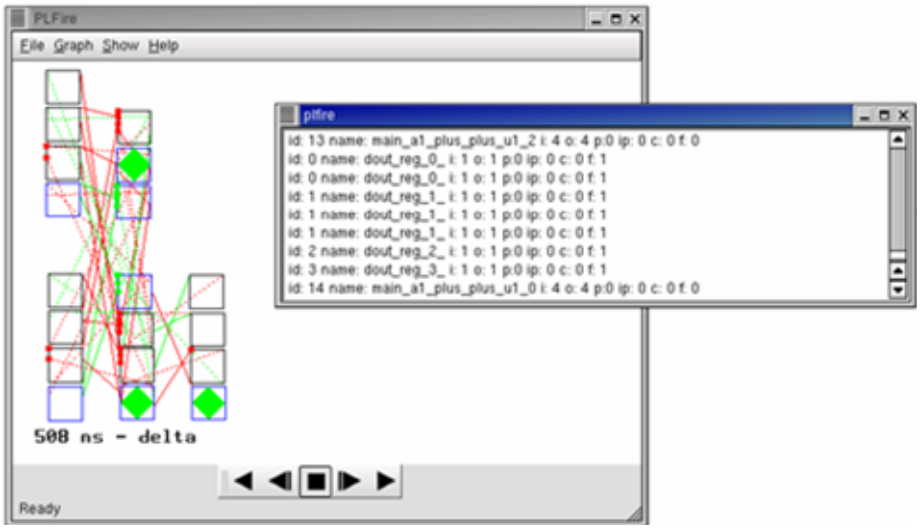


Figure 6.1. PLFire Screen Shot

design. The `viznet` file is generated by the mapper that takes a synchronous circuit in `EDIF` format and converts it to a PL circuit in `VHDL` format. The `viztiming` file contains the firing information of a PL design. The `viztiming` file is generated when the `VHDL` file is simulated.

6.2. Implementation

`PLFire` is written in `C/C++` and uses the `QT GUI libraries` and `OpenGL`. `QT` is a free set of libraries that allows a programmer to create *graphical user interface* (GUI) programs for the `UNIX/Windows/Macintosh` platforms. `OpenGL` is a free, standardized set of graphics libraries that many platforms support.

The current design environments are `Sun Solaris` using `Qt 3.0.1` and `OpenGL` libraries. Also, `Microsoft VC++ 6.0` is used for additional testing under the Windows environment.

The tool can be broken into four categories:

1. Graphical User Interface

2. PL data structures

3. Firing data structure

4. PL Display Code

6.2.1. Graphical User Interface

The GUI consists of a main window that contains menus for loading data other function and a display for the PL circuit. The `QT` libraries handle the internals of the main window. Programming consists of mainly declaring buttons and other interface structures and defining their functions. PL Data Structures

A set of classes to represent the various aspects of a PL circuit was implemented. Using object oriented design techniques, data structures were implemented with the mindset that components should be easily modified and replaceable, and that future components and enhancements could be easily implemented. To store the netlist information three classes are used.

1. PL System stores high level information about the PL circuit including lists of gates, signals, and firing information

2. PL Gate stores general information about PL gate instances and provides functions for PL Gates

3. PL Signal stores the state PL signal instances and provides functions for a PL Signal

Note, a PL Signal represents each fanout from an output terminal. This is true about input terminal fan-ins as well. There is also a PL Element parent class (PL Gate and PL Signal inherit this class). This is so that future PL components can be easily implemented. The PL netlist data is loaded from a `viznet` file generated by the mapper tool and placed into a PL System instance.

For each gate and signal read, `PLFire` will construct a PL Gate or Signal class instance. Each instance is then pointed to by an array of their respective type. This array of pointer structure for PL Gate and PL Signal lists allows for quick retrieval of the instance by the program.

6.2.2. Firing Information Data Structure

Firing information is loaded into a `PLFire` Element structure. The main components of the structure hold the following information:

1. Time of firing
2. Gate to be fired

Each firing is loaded into a doubly linked list of other firings that occur at the same time. This list of common firings is then loaded into a master list of firings, which is also a doubly linked list. This master list is referred to as the To Be Fired (`2BF`) List. The order of the master list is in chronological order. In addition, the CFL maintains dummy head and tail elements (which contain firing times of "never") to allow for animation in forward and reverse time. Also, this allows for better data access for the display algorithm, which will be discussed next.

6.2.3. PL Display Code

There are 2 main aspects of the display code.

1. Show the components and tokens
2. Animate the components

Displaying the components consists of passing a pointer of the component to a DisplayComponent Function, which in turn places the component depending on its type. `OpenGL` is then used to draw the components to the screen.

To animate the netlist, the `2BF` is processed. Also, a list of signals that have fired is maintained. The contents of the list are initially signals with initial tokens,

```
SList = signals that have previously fired
2BF = list of current firings

Animate()
{
        Signals_2BF = list of signals to be fired

        If(2BF.firing_time = now)
                Consume_Tokens(SList)
                For each 2BF.gate
                        handleFiring()          //toggle phace of gate and
                                                //its output signals
                        Signals_2BF = Signals_2BF ∪ outputSignals
                         sList = (sList - inputSignals) ∪ Signals_2BF
                        Animate_Tokens(Signals_2BF)
                        2BF = 2BF.next
        Else
                 Do Nothing
}
```

Figure 6.2. Pseudocode of Animate()

since no signal can fire before hand. This information is inherent in the PL Firing List through its gate elements, but having this separate list makes the animation of tokens more efficient, i.e.: bypasses the gate element reference.

This algorithm works as follows. `Animate()` will check if the current time is when the next set of firings needs to happen. If yes, then change the state of the PL system and animate token flow. This algorithm shows how forward animation is handled for a forward flow of time. A similar algorithm is used for "rewinding" the token flow.

Chapter 7

CONCLUSION

In this thesis, a presentation of automation techniques for speed up mechanisms in Phased Logic was given. We have seen that the truth table approach is a viable option for fine-grained PL, as the functions involved are of minimal size. The experimentation results show that EE can improve circuits, particularly circuits such as processors, by as much as 40%. Furthermore, implementations for automatically inserting EE for larger circuits, were investigated and can be readily implemented for coarse-grained PL. A description of an automatic slack matching buffer insertion algorithm was given as well. Although the implementation has the undesirable property of using a simulator, it is adequate to give hints as to where buffers may be inserted to improve performance. Additionally, a visualization tool for token flow through a PL netlist is presented.

Ideas for future research for improving the topics presented are shown below:

- Early Evaluation
  - Better Heuristics for selection of $\underline{A}$
  - Better analysis tools for determining timing characteristics of master function inputs
  - Application of EE in other areas of asynchronous or synchronous design
- Slack Matching Buffering
  - Techniques for slack matching buffering that does not rely on simulation

- Phased Logic
  - Verification of PL design with and without the optimization techniques described
  - Testing of PL circuitry

REFERENCES

[1] Berkel, K. v. Beware of isochronic fork. *Integration: the VLSI journal 13* (June 1992), 103–128.

[2] Brayton, R., Hatchel, C., and Sangiovanni-Vincentelli, A. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[3] Bryant, R. Graph-based algorithms for boolean function manipulation. *IEEE Transations on Computers 35*, 8 (1986), 677–691.

[4] Bryant, R. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers 40*, 2 (February 1991), 205–213.

[5] Commoner, F., Hol, A., and Pneuli, A. Marked directed graphs. *J. Computer and System Sciences 5* (1971), 511–523.

[6] Dean, M., Dill, D., and Horowitz, M. Self-timed logic using current-sensing completion detection (cscd). *Journal of VLSI Processing* (July 1994).

[7] Dean, M., Williams, T., and Dill, D. Efficient self-timing with level-encoded 2-phase dual-rail (ledr). *Advanced Research in VLSI* (1991), 55–70.

[8] Itc99 benchmark set - politecnio di torino. http://www.cad.polito.it/tools/itc99.html, 1999.

[9] Fazel, K., Li, L., Thornton, M., Reese, R., and Traver, C. Throughput enhancement in phased logic circuits using automatic slack matching buffer insertion. In *ACM/IEEE Great Lakes Symposium on VLSI (GLSVLSI* (April 2004).

[10] Fazel, K., Thornton, M., and Reese, R. Plfire: A visualization tool for asynchronous phased logic designs. In *IEEE/ACM Conference on Design, Automation, and Test in Europe (DATE)* (Munich, Germany, March 2003), pp. 1096–1097.

[11] Gao, M., Jiang, J.-H., Jiang, Y., Y., L., S., S., and R., B. Mvsis. In the Notes of the International Workshop on Logic Synthesis, June 2001.

[12] Harris, D. *Skew-Tolerant Circuit Design*, 1st ed. Morgan KAufmann Publishers, San Francisco, CA, 2001.

[13] Hauck, S. Asynchronous design methodologies: An overview. *Proceedings of the IEEE 83*, 1 (January 1995), 69–93.

[14] Huffman, D. Design of hazard-free switching circuits. *Journal of the ACM 4* (January 1957), 47–62.

[15] Kukimoto, Y. *BLIF-MV*. The VIS Group, University of California, Berkeley, May 1996.

[16] Linder, D. *Phased Logic: A Design Methodology for Delay Insensitive Synchronous Circuitry*. PhD thesis, Mississippi State University, 1994.

[17] Linder, D., and Harden, J. Phased logic: Supporting the synchronous design paradigm with delay-insensitive circuitry. *IEEE Transactions on Computers 45*, 9 (September 1996), 1031–1044.

[18] Lines, A. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1995.

[19] Martin, A. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the 1990 MIT Conference on Advanced Research in VLSI* (1990), pp. 263–278.

[20] Meinke, T., Hemani, A., Ellervee, P., Öberg, J., Kumar, S., Lindqvist, D., Tenhunen, H., and Postula, A. Evaluating benefits of globally asynchronous locally synchronous vlsi architecture. In *Proceedings of the 16th IEEE NORCHIP Conference* (Lund, Sweden, November 1998), pp. 50–57.

[21] Moore, S., Anderson, R., Cunningham, P., Mullins, R., and Taylor, G. Improving smart card security using self-timed circuits. In *Eighth International Symposium on Advanced Research in Asynchronous Circuits and Systems* (2002).

[22] Muller, D., and Banrtky, W. A theory of asynchronous circuits. In *Proc. Int. Symp. of Theory of Switching* (1959), vol. 29, pp. 204–243.

[23] Murata, T. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE 77*, 4 (Apr. 1989), 541–580.

[24] Paver, N., and Edwards, D. Is asynchronous logic good for low-power? In *IEE Colloquim on Low Power Analogue and Digital VLSI: ASICS, Techniques and Application* (June 1995).

[25] Petri net tools database quick overview. http://www.daimi.au.dk/PetriNets/tools/quick.html, 2004.

[26] Reese, R., Thornton, M., and Traver, C. Arithmetic logic circuits using self-timed bit-level dataflow and early evaluation. In *Proc. ICCCD* (Austin, September 2001), pp. 18–23.

[27] Reese, R., Thornton, M., and Traver, C. A coarse-grain phased logic cpu. In *IEEE International Symposium on Asynchronous Circuits & Systems* (May 2003), pp. 2–13.

[28] Reese, R., Thornton, M., and Traver, C. A fine-grain phased logic cpu. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (February 2003), pp. 70–79.

[29] Reese, R., and Traver, C. Synthesis and simulation of phased logic systems. In *International Workshop on Logic Synthesis* (Dana Point, California, 2000), pp. 255–259.

[30] Rodriguez-Villegas, E., Huertas, G., Avedillo, M., Quintana, J., and Rueda, A. A practical floating-gate muller-c element using vmos threshold gates. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on 48*, 1 (January 2001), 102–106.

[31] Seitz, C. System timing. In *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Addison-Wesley, 1980.

[32] Somenzi, F. Colorado university decision diagram (cudd) package. http://vlsi.colorado.edu/ fabio/CUDD, March 2004.

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

[34] Sutherland, I. Computers without clocks. *Scientific American* (August 2002), 62–69.

[35] Teich, J., Sriram, S., Thiele, L., and Martin, M. Performance analysis of mixed asynchronous synchronous systems. In *IEEE Workshop on VLSI Signal Processing* (1994).

[36] Thornton, M., Fazel, K., Reese, R., and Traver, C. Generalized early evaluation in self-timed circuits. In *Proc. Design, Automation, and Test in Europe* (Paris, France, March 2002), pp. 255–259.

[37] University of California, Berkeley. *Berkeley Logic Interchange Format BLIF*, December 1998.