CLUSTERING AND FANOUT OPTIMIZATIONS OF ASYNCHRONOUS CIRCUITS

by

Georgios D. Dimou

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(ELECTRICAL ENGINEERING)

May 2009

Dedication

To my family

Acknowledgements

I would like to thank my advisor Professor Peter A. Beerel for his valuable guidance and support though the entire course of this work. I would also like to thank Professors Massoud Pedram and Ken Alexander for taking the time to be part of my defense committee.

I would also like to thank TrellisWare Technologies Inc. for their support and flexibility they have provided me with throughout my employment with them, which allowed me to complete this effort.

Additionally I would like to thank Fulcrum Microsystems for providing assistance in this effort and specifically Andrew Lines for his input and the discussions that we had during the meetings between our USC team and Fulcrum Microsystems.

I would also like to specifically acknowledge my friend and colleague Sotirios Zogopoulos for proof reading my work and for the exchange of ideas that we had in the process as well as Mallika Prakash, Arash Saifhashemi and Pankaj Golani for their help in developing parts of the flow that the testing platform for this thesis was built on. I would also like to acknowledge Nam-Hoon Kim for the code-base that he developed, which I was able to build upon in order to test this work.

I would also like to thank my girlfriend Kristen Grande for her emotional support through the years.

Finally and most importantly I would like to thank my parents Dimitrios and Evanthia-Maria Dimou for enabling me and supporting me throughout my education that lead up to this thesis and for being there for me any time I needed help or advice. Without them this work would have never been possible.

Table of Contents

List of Tables

List of Figures

Abstract

As semiconductor technology advances into smaller and smaller geometries, new challenges arise. The increased circuit integration combined with larger variability make it harder for designers to distribute a global clock and global interconnect signals efficiently in their designs. To combat the effects designers use more conservative models and more complicated tools that result in longer design times and diminishing returns from the migration to the smaller geometries. Some of these problems can be addressed by asynchronous circuits, but there exists no well-defined method for automated asynchronous design. Some methods have been proposed over the years, but they leverage off existing synchronous techniques too much, resulting in circuits that are bound by the characteristics of their synchronous counterparts. This thesis proposes a novel approach for generating such circuits, from any arbitrary HDL representation of a circuit by automatically clustering the synthesized gates into pipeline stages that are then slack-matched to meet performance goals while minimizing area. The method thus provides a form of automatic pipelining in which the throughput of the overall design is not limited to the clock frequency or the level of pipelining in the original RTL specification. Consequently, the performance can generally maintain or exceed the performance of the original circuit. The method is design-style agnostic and is thus applicable to many asynchronous design styles.

The contributions of the thesis are two-fold. First, we define a model and theoretical infrastructure that guides clustering to avoid the introduction of deadlocks and achieve a target circuit performance. This provides a framework for proper clustering that can enable the unhindered exploration of area minimization algorithms in the future and lead to optimized competitive designs. Secondly, we propose optimizations to existing slack matching models that take advantage of fanout optimizations of buffer trees that improve the quality of the results.

Chapter 1 Introduction

Synchronous design has dominated the VLSI and ASIC design market for many years. Recently the design challenges of designing circuits with a global clock has been rising, as the industry moves to smaller geometries that allow for very large circuits to be placed on a single die. In particular, as wiring delays become more and more dominant, the task of distributing a clock across a large design and dealing with clock delay and skew across the entire die is becoming a growing problem. The usual trend of clock frequencies in commercial chips increasing every year has stopped and even though manufacturers are moving to smaller geometries clock frequencies have remained constant for some time.

At the same time, higher chip integration is also introducing problems in terms of power consumption, much of which is coming from the clock distribution circuits. Processor manufacturers are resorting to more complex implementations of clocking strategies involving clock gating, software-controlled clock frequencies and voltages across the chip, such as PowerNow! of AMD [2] and Intel's Enhanced SpeedStep Technology [1]. Both technologies are used to reduce power consumption when the processors are moderately used or idle.

Such problems are inherently solved in asynchronous circuits. The elimination of the global clock removes all the complex timing requirements associated with its distribution. It also allows the designer to remove timing margins built into the logic to accommodate clock variations, and use this time for processing. Power in asynchronous circuits is also only consumed during processing as the control circuits are data-driven. On top of these benefits additional area becomes available on the die due to the elimination of the clock generation and distribution network. Several asynchronous design styles have been developed that span a variety of application domains, from low power to high-performance.

Although asynchronous logic is incredibly appealing, it has yet to be adapted widely by the semiconductor industry. Until recently asynchronous design was synonymous to custom design, since no automated flow existed for the generation of asynchronous circuits. For the majority of the commercial world, where ASIC flows are the typical approach for chip design, this is a major drawback, since most companies lack the infrastructure and the expertise to support such efforts. The cost and the time-to-market for such efforts are usually prohibitive for companies that do not specialize in the area of custom VLSI design. On top of that, critical issues such as verification and testing, which is normally done by commercial tools is not supported for the majority of design styles that are proposed in the asynchronous logic research community.

The advantages of asynchronous design, combined with the lack of a formal automated flow for the generation of such circuits are the key drivers behind this work. The goal is to examine existing asynchronous design styles and identify key problems that are unique to asynchronous design. We want to characterize those problems and formulate their solutions, so that a tool can be generated to address those asynchronous-specific issues. Having performed that step, we can then leverage off commercial tools and solutions used in current synchronous design flows, and complete a full set of tools and procedures that would allow a designer to generate an asynchronous design from behavioral RTL code. It is our belief that the existence of such a flow would open the way for a wider adaptation of asynchronous logic and a realization of its benefits in the commercial world.

1.1 Contributions

The main focus of this thesis is in the area of automated pipelining of asynchronous circuits. The method proposed here involves starting from a gate-level representation of a circuit, in which each gate represents a pipeline stage, and merging two pipeline stages at a time to generate larger pipeline stages. This method allows us to generate a pipelined circuit from an arbitrary netlist that can take advantage of performance and design features of various asynchronous design templates. In order for this process to be meaningful these merging transformations have to be designed based on two important criteria. First the transformations have to be guaranteed not to cause a circuit to become non-functional. Secondly the clustering transformation ensures that after slack matching the target circuit performance can be achieved.

We model the circuit as a directed graph to abstract the implementation details of the circuit. Clustering is then the iterative merging of nodes of the graph. Based on this model this thesis includes the following contributions:

- Demonstrates criteria that help detect and prevent merges that could generate a transformed graph that cause deadlock. The deadlock-free maintenance criterion is expressed in two different ways for practical reasons. The first expression uses a breadth-first search in the graph while the other depends on the existence of all pair-wise distances in the graph, which is a computationally intensive process. Both have the same worst-case complexity, but the search-based one has better average-case performance and in practice is proven to be much faster. However the distance-based algorithm has the advantage that it can be seamlessly combined with criteria that help ensure target performance.

- Proposes additional criteria that help detect and prevent graph transformations that could introduce logic structures that would reduce the performance of the circuit beyond the target performance that is required for a particular design. This means that while clustering these criteria will prevent the artificial lengthening of the critical paths of the design beyond the point where it is impossible to satisfy the cycle time or the end-to-end latency constraints for the design.

In the process of developing this main framework, other contributions were made that will have an impact on the general asynchronous design process and in particular in the area of RTL-based (ASIC) flow.

- Developed local update for the all pair-wise distance array that is generated by a modified Floyd-Warshall algorithm so that the array can be updated with our clustering transformations with minimal local updates reducing complexity significantly.

- Developed practical improvement to the Slack Matching formulation originally presented in [5] to incorporate additional information for potential buffer sharing and thus improved the end result.

In order to test and verify all the ideas that were developed as part of this thesis a software package was developed that enables an asynchronous ASIC flow. The software implements an end-to-end tool that reads a Verilog netlist and performs the operations described in this thesis.

- Currently reads in Verilog netlists that are gate-level synthesized netlists from any commercial software package. It recognizes the MLD and PCHB design styles currently and can translate to fully function netlists for either design styles.
- Performs clustering on a graph which is design-style-agnostic and can use all different variants of the clustering as described in Chapter 3 and.Chapter 5.
- It also performs slack matching by utilizing external linear programming solvers and fanout checking (and fixing if necessary) on the netlist to guarantee that all design rules are followed.
- Emits Verilog netlists that can be used as input to the back-end tools (applicable to any software package available) as well as testbenches for both the original and final netlists and random vector generation code that allows pre- and post-software behavioral verification.

1.2 Outline

In order to better describe the nature of the problems that will be discussed as part of this work, Chapter 2 has been devoted to summarize the various existing asynchronous design styles, as well as existing synthesis flows, their features and weaknesses. Some key challenges and definitions are also defined here to present the general context for this research. Chapter 3 presents a thorough analysis of the clustering theory for asynchronous circuits that was developed and how it can be used to guarantee functionality and performance. Some interesting contributions to other flow issues that are also involved in the end-to-end design process, namely slack matching and fanout optimizations, are presented in Chapter 4. Chapter 5 presents the existing version of our tool and flow, its features, capabilities and inefficiencies. Chapter 6 summarizes the conclusions of this work and identifies interesting problems for future work in the area.

Chapter 2 Background

2.1 Introduction

In synchronous circuits all the data is synchronized by the global clock. In between combinational logic blocks latches or flip-flops are inserted that latch the data once per period of the clock hence achieving the synchronization of data and control signals among the different circuit elements. In asynchronous circuits this synchronization is achieved through handshaking protocols that are implemented to assist the various circuit elements with the exchange of data. There are many styles of asynchronous design libraries and flows, and almost each one has a different handshaking mechanism associated with it. Before attempting to go into detail describing different asynchronous design styles it is useful to go through the various classifications of asynchronous protocols and their properties so that it is easier to place the different options as part of a bigger picture. The classification can be done in many dimensions, namely based on the handshaking interface, data interface and the timing requirements that have to be satisfied for correct operation of the circuit.

2.2 Handshaking protocols

The handshaking between two asynchronous units exchanging data often starts with the unit where the data is originating from sending a request to the receiver. Typically the request is sent when the data is ready and depending on the protocol this could be part of the data or a separate control signal. The receiver has to then acknowledge the receipt of the data. Then the transmitting module knows that the data has been consumed and can reset its value, in order to be ready to process the next set of data. This Request-Acknowledgement exchange can be performed in several different ways and handshaking protocols can be classified according to the nature of this exchange.

First, we distinguish between protocols according to the number of phases (transitions) that exist in the handshake. There are two distinct kinds of protocols, the 2-phase and the 4-phase protocol. In the 4-phase protocol case the sender asserts its request (REQ) to inform the receiving element that it holds valid data on its output. The receiving element will then receive the data when it is ready to consume it and raise the acknowledgment (ACK) signal when it has actually done so. The sender will then reset its REQ signal and after that the receiver will lower its ACK signal. The second pair of transitions could also be used to explicitly identify a data reset phase. The 2-phase protocol only uses two active transitions to complete the communication handshake. Therefore, all transitions of the REQ/ACK signals are used in the same way, whether falling or rising. That means that during the first cycle the sender raises REQ and then the receiver raises ACK to finish the handshake. Instead of resetting the signals before the second communication, the protocol is implemented so that the sender lowers REQ to start the next transfer, and then the receiver lowers ACK to acknowledge the data. The request and acknowledgment signals could be individual signals or they could be implemented across the same wire. The later is also known as single-track communication, and could be implemented by having the sender assert the signal (REQ) and the receiver de-asserting it. These three cases are summarized in Figure 1.

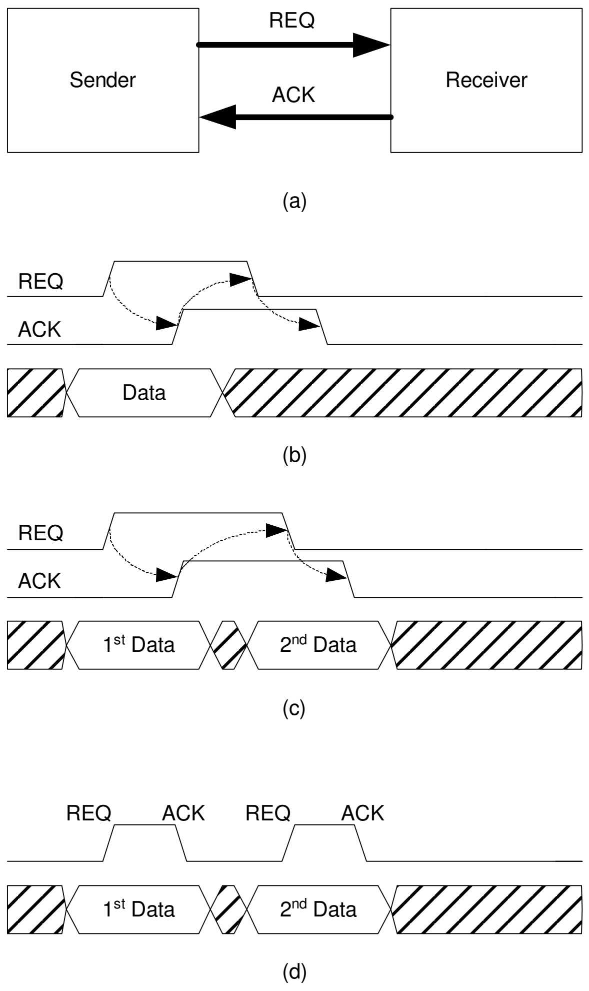


Figure 1 **Summary of asynchronous handshaking protocols based on the messages that are exchanged between sender and receiver. Based on this classification we identify 4-phase protocols (b), 2-phase protocols (c) and single-track 2-phase protocols.**

Another interesting point is that the basic protocols described above could be modified by interleaving different edges to serve different purposes. The basic forms described above are for point-to-point communications between two adjacent units and the communication cycle is always initiated by the sender. When the sender initiates the protocol it is considered a *push* channel, and they are common in pipelined circuits. However, in other non-pipelined circuits the receiver signals that it is ready first before the sender produces any data. This is known as a *pull* channel and the initial request is sent by the receiver and in the reverse direction of the data flow. For example an adaptation of the 4-phase protocol described previously for *push* channels can be used for *pull* channel communications. The receiver asserts the REQ signal to indicate that it is ready to accept data. When the sender has computed the data and put it on the channel it asserts its ACK signal. The receiver then lowers its REQ signal as soon as it has consumed the data. Finally the sender lowers its ACK signal after it has reset the data and the channel is now ready for the next transmition. This is depicted in Figure 2.

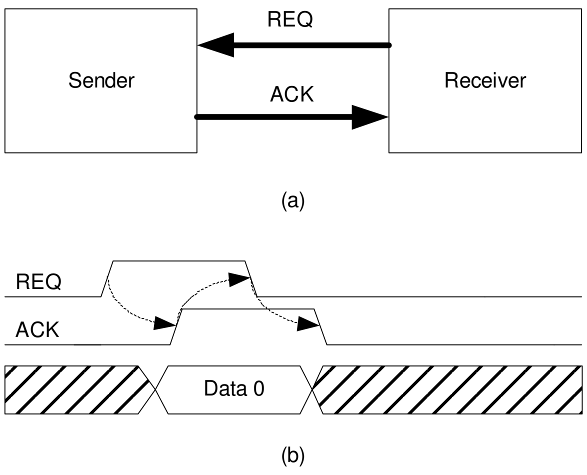


Figure 2 An example of a 4-phase pull channel.

All the examples stated up to this point are examples of point-to-point communications. This means that the sender sends a signal to indicate the presence of data and releases the data when that gets acknowledged. Another quite interesting case is called *enclosed communication*. It is defined as the case where the REQ signal is asserted and then followed by an entire handshake from the receiver side (meaning the ACK is both asserted and de-asserted), before the REQ signal gets de-asserted. This type of behavior might not make a difference in a typical *push* pipelined channel, however its usefulness becomes apparent when considering cases where we want to perform sequential actions instead of concurrent actions. Assume that the sender wants to generate data and then there are multiple receivers that are going to operate sequential actions based on this data. The REQ signal can then be asserted to validate the data on the sender side. Then multiple receivers can take turns operating on the data and the REQ signals stays high validating its presence. When the last one of the receivers is done processing the sender can lower the REQ signal and reset the data. The signal transitions for such a scenario are shown in Figure 3. Additionally it can also be the case that some or all of these processes operate on the data with some level of concurrency as well.

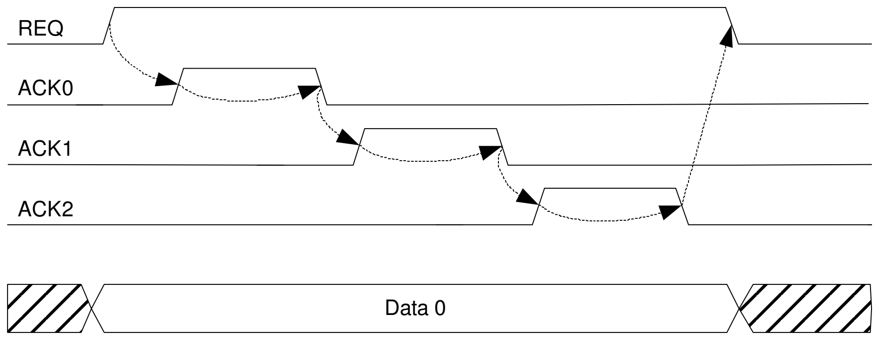


Figure 3 An example of an enclosed interleaving of four channels with one sender and three receivers.

2.3 Data Encoding

Another way of classifying asynchronous channels is based on the way that the data is encoded on the channel. The way that is closest to typical synchronous designs is called bundled data. In bundled data the data is presented in the form of a bus of single rail wires from the sender to the receiver. This has the benefit that only one wire per signal is only required and that the signals could be generated by single-rail combinational blocks just like those used for synchronous design. However there is no way to identify that the data is valid on the receiver end by just observing the data rails, hence the designer has to make sure that the data is all valid before the REQ signal becomes visible to the receiver. For this reason the REQ path has to be delay matched with the slowest combinational path between sender and receiver and this task is not trivial. Post layout simulation is typically required to ensure the functionality of the circuit.

Another way to encode data on a channel is by making it dual-rail. If the dual-rail signals are reset between transitions it is now easy to verify the presence of the data by the data itself by making sure that at least one of the two wires representing the data has been asserted. In this case an explicit REQ line is not necessary for the data, as a simple OR of the two signals verifies that the data is present. Dual-rail signals can also be grouped together in busses as in bundled data. If there is no explicit REQ like in the bundled-data rails all the individual OR results from each signal has to be combined to generate the global REQ signal for the bus. When one bit is transferred a single gate delay is added to the critical path, but in the later case the impact of such a circuit to the performance of the circuit could be significant since it could amount to several gate delays.

Finally a more generalized for of dual-rail signaling is 1-of-N signaling. Here for every *n* wires that are used one can transmit *log(n)* bits. Out of the *n* wires only one is asserted at a time. This encoding has several benefits. Just like dual rail signaling there is no need for an explicit REQ signal since the presence of data can be extracted from the data itself (again assuming that the data is reset between transmitions). It has also been shown in [17][18][20] that, with this type of data encoding, one could also omit the acknowledgement signal as well. Moreover, there is less activity on the bus than in the dual-rail case, since only one wire is allowed to switch during a transmition, which could help reduce power consumption and crosstalk noise. However the width of the bus grows exponentially with the amount of bits to be encoded and therefore, this approach could quickly become impractical for the implementation of wide buses. For such wide data paths the signals have to be broken up into smaller groups. Popular solutions are 1-of-2 and 1-of-4 encoding since they use only 2 wires per signal, but 1-of-8 is also common due to better power efficiency.

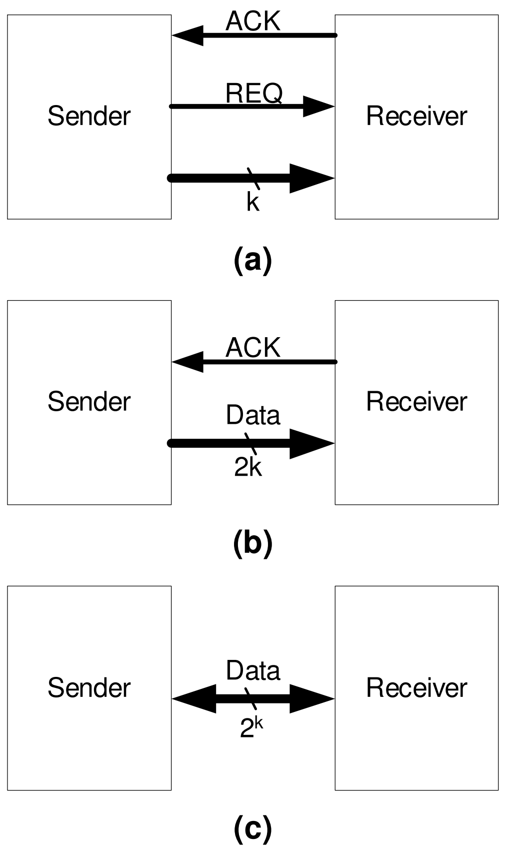


Figure 4 **Abstract view of different channels that transmit k bits of data; (a) using the bundled-data protocol, (b) a differential bus and (c) single track 1-of-N communication.**

2.4 Timing Assumptions

One last classifying characteristic of asynchronous communication channels is the type of timing assumptions that are required to hold for a particular protocol to operate correctly. In terms of the actual design process, the fewer timing assumptions that exist in a design the better, since timing assumptions usually have to verified through simulation that have to be performed both pre- and post-layout.

The first timing model is one that all delays both gate and wire are allowed to assume any value, and the circuit is guaranteed to function properly. This model is called *delay insensitive* (DI), and it is the most robust model for asynchronous circuits. However, Martin [26] showed that no practical single-output gate implementation exists that satisfies this constraint. Realistically this means that in order to build a DI circuit one would have to use only multi-output gates, which is incredibly restrictive for typical circuit designs. Alternatively one would have to use gates that cannot be designed according to the DI timing assumptions and integrate them in the circuit using DI rules at the top level.

Martin in [27] also proposed an alternative to this strict constraint called *Quasi Delay Insensitive*. This has the same delay assumptions as the DI except it requires that every fork in the design is *isochronic*. The *isochronic* fork is a fork, for which the delays to all destinations out of that fork are equal. This realistically is very hard to achieve due to varying wire lengths and load at the destination during design, and due to varying operating conditions (such as voltage and crosstalk noise) during actual operation of the circuit. More realistically this assumption can be approximated by constraining the short path departing from a fork to be faster than slowest reconvergent path to the gate driven by the short path. This is a constraint that is much easier to meet and verify in practice. In [3] the QDI concept is extended from short wire fork delays to delays through a small number of gates that depart from a fork and then re-converge further down the data path.

Another category of circuits are *Speed-Independent* circuits (SI). In speed independent circuits gates could have arbitrary delays, but wire delays are considered negligible. This makes all forks *isochronic*, hence the QDI protocol requirement stands by default. With process geometries constantly shrinking though, wire delays become more and more dominant part of a path delay, and this assumption and the real delays need to be determined post-layout and the functionality of the circuit has to be verified again through simulation.

Scalable Delay Insensitive (SDI) is an approach that partitions the design in smaller parts and attempts to bridge the gap between DI and SI through this approach. Within each sub-module the design is performed by bounding the ratio of delays between paths by a constant. It also defines a ratio related to the estimated and observed data on the delays that is also lower and upper bound. The same constant is used as a bound for both expressions. After each individual module is designed, the interconnections at the top level are designed based on DI assumptions.

2.5 Asynchronous Design Styles

As one can see there are many design choices available, both in terms of the handshaking protocol and data encoding. Each has different advantages that could be exploited depending on the requirements of a project. Research teams have over time defined many design styles that can be used for the implementation of asynchronous circuits, each one having its own advantages and disadvantages. These styles are collections of design libraries, protocol definitions and constraints that have been designed and verified to produce functional designs. These basic cells in these libraries can be used to produce the desired circuits, and this flow, at least theoretically, can be either manual though a custom VLSI approach or automated just like a synchronous ASIC flow. In the process of defining the context of the problems that this work is attempting to solve, it is important to go through a few popular and proven design styles that have been developed over the years and proven their value for certain types of design requirements.

2.5.1 PCFB and PCHB

The Pre-Charge Half Buffer (PCHB) and Pre-Charge Full Buffer (PCFB) were presented in [7] and are two example of a QDI template. Both templates are similar, but PCFB uses an extra internal state variable so that it is able to store one token per stage, and that is why it is called a Full Buffer. On the other hand a PCHB is a half buffer meaning that one token can exist in two adjacent pipeline stages. The templates are designed for fine-grain-pipelining, which implies that each pipeline stage is one gate deep. The data is encoded using 1-of-N encoding and thus there is no explicit request line associated with the data. Each gate has an input completion detection unit and the output also has an output completion detection unit. The two templates are shown in Figure 5.

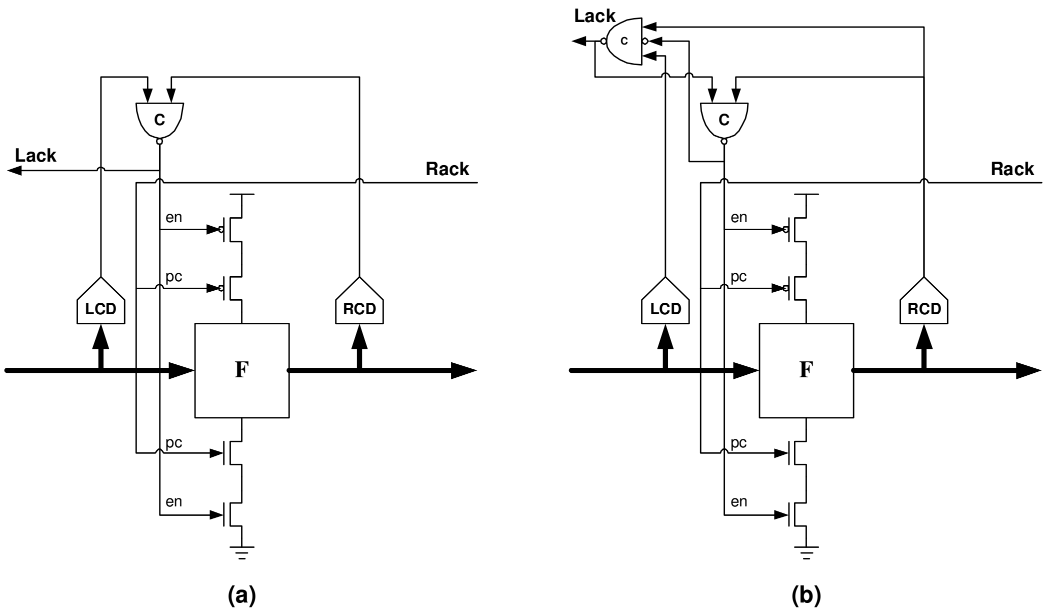


Figure 5 The PCHB template is shown in (a) and the PCFB template is shown in (b).

The function blocks are designed using dynamic logic (Domino logic [32] more specifically) in order to reduce the size of the circuit. Another interesting property is that the function block can actually evaluate even if not all inputs are present yet. If the function allows it the function block can generate an output with a subset of the inputs and data can propagate forward along the pipeline. However the C-element will not send an acknowledgement to the left environment until all inputs arrive and the output has been generated. That prevents premature acknowledgments from propagating backwards to units that have not even produced data yet. The RCD is used to detect that data has indeed been generated from the function block. In the PCHB when both the LCD and RCD have detected valid data on both input and output the function block gets disabled. When the next stage in the pipeline acknowledges the outputs of the current stage then the function block will be pre-charged to be ready to receive the next set of data.

The LCD and RCD operate on 1-of-n encoded channels. Their operation is performed simply by performing an OR on the two wires. The data is reset to zero during pre-charge, therefore, the presence of data is detected when one of the two wires produces a logic 1. If multiple channels exist the results of the OR from each channel have to be combined together through C-elements to produce the output of the LCD/RCD. Even though this is a simple operation one has to remember that this a fine-grain-pipeline design style. For multi-input gates the control logic quickly becomes a large overhead and as a result these templates are not area efficient. Also even though the cells use dynamic logic for smaller size and better performance, there are several levels of control involved in the critical path. With PCHB being a half-buffer the cycle time involves multiple levels of logic as well as a completion detection unit and a C-element. Its cycle time varies depending on the functional block, but is generally between 14 & 18 transitions. The PCFB is a full buffer version of PCHB. It has the same cycle time as PCHB, so its only benefit would be slack capacity. For this reason the PCFB is not as widely used as the PCHB design style. Even though this yields good overall performance, there are design styles available that have much smaller cycle times.

2.5.2 Mousetrap

MOUSETRAP is a recently proposed design style [38] that seems very interesting due to its versatility and speed. It is a bundled-data protocol, with 2-phase control and could be used for both very fine-grain and coarser pipeline design. It has a very small cycle time of 5 transitions for a FIFO design and although the cycle time would increase with merges, fork and logic added to it, it still has the potential for very high throughput implementations. A basic diagram of a FIFO pipeline designed with this design style is shown in Figure 6.

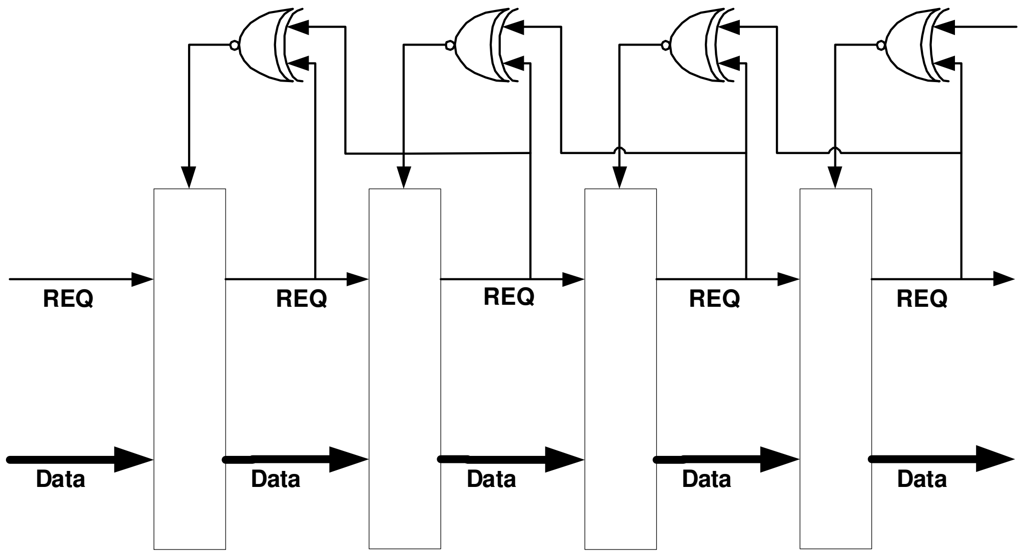


Figure 6 The MOUSETRAP design style for a FIFO implementation.

The most interesting feature of this type of circuit is the simplicity of its control. The same feature is to a large extent responsible also for its small cycle time. It essentially consists of a single XNOR gate per pipeline stage. The data is accompanied by a request that has alternating values for successive tokens. The data and the request are both latched at every stage with latches that are controlled by the local control (XNOR). The latches are initially transparent waiting for data to arrive. When data goes through, since the requests of successive token have alternating values, when the request goes through the latch it will change the value of the XNOR output and the latch is made opaque. When the next stage has also fired and the data has been latched there the XNOR will change value again and the latch is made transparent again and new data may arrive. If the design is not just a FIFO, then logic is added on the data path between the latches. In that case since the data is single-rail bundled data, the request has to be delay-matched with the slowest combinational path in order to avoid errors in the interface. When the request goes through is will go through the XNOR and close the latch, so the timing has to be designed so that the data will definitely be through the latch before that happens.

One big advantage of this design style is that the data path can be designed from standard gates that can be found in any synchronous library. That means that no custom asynchronous cell library design is necessary to support the protocol and also that it might be easier to use existing tools for an automated synthesis flow. However, this style requires the use of delay-matched request lines and has one timing assumption that needs to be verified for every pipeline stage. These generally require more cumbersome verification both pre- and post-layout. The idea has been presented for FIFOs, however it could be extended to more complex pipelines with forks, joins and cycles. In those cases the fact that successive tokens require requests with alternating values becomes a restriction that needs to be designed with care. Also in the case of merges and even more specifically conditional merges it is our assessment that more complex cells with memory of the previous state are required to handle such a pipeline.

2.5.3 MLD

Multi-Level Domino is another design style that also used bundles of wires, however here the data is encoded using differential encoding. The data path is constructed out of domino-logic gates in order to be more area efficient as well as faster. This also allows the circuit to generate a request to the next stage based on the data itself. A completion detection unit exists for each output and all the validity signals are then combined through an AND gate tree to generate the valid flag for the entire pipeline stage. The style is targeted more towards medium-grain pipelining and several layers of logic and many data paths in parallel are typically used in a single pipeline stage. This yields a small overhead from the addition of the pipeline stage control units and hence an area efficient design. Several variations of this design style have been proposed over the years in like PS0 [39] [42], and LDA [7], others using 2-phase and others using 4-phase handshaking. Even though there are differences between the variants in terms of the handshaking mechanism of the controllers and the generation of control signals, abstractly the general form of these styles can be illustrated in Figure 7.

The cycle of a pipeline stage starts with the dynamic logic gates receiving data from the previous stage and evaluating their outputs. When the data propagates to the last stage of gates in the pipeline stage the outputs for the stage are generated and the dual-rail signals are used to validate that all outputs are present. The valid signal is generated for the entire stage and is used as a request to the next stage. It could also be used internally in the stage for isolating the outputs and initiating an early pre-charge of the logic before the final stage. When the next stage acknowledges the data, the stage resets its outputs to all zero so that the valid signal is forced low. The data path is connected normally just as in the case of a synchronous netlist. Any forking or merging between stages is handled by the controller circuits. That can be accomplished by inserting C-elements for the requests of signals reaching a merge and the acknowledgment signals departing a fork. The introduction of such elements might impact the cycle time of a stage, but since the data path is several stage long, this extra delay can be offset by reducing the amount of logic levels in a particular stage.

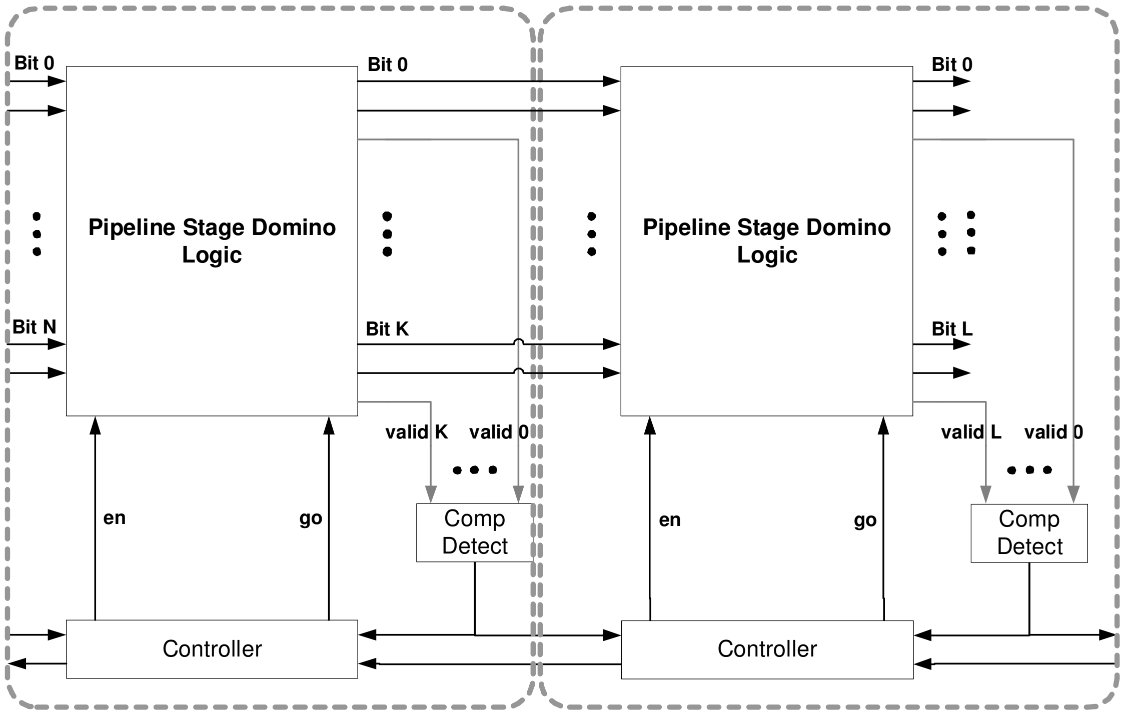


Figure 7 An abstract view of a Multi-Level Domino Pipeline.

The generation of the request from the data alleviates the need for delay lines that are required in single-rail data paths such as MOUSETRAP. It also simplifies the timing verification required for the designs. On the other hand since dual-rail dynamic gates are not available commercially this style requires that libraries are generated specifically for this application and this increases the design time and makes automated synthesis harder since commercial synthesis tools are not currently capable of handling dual-rail signaling. Another interesting point of the design style is that since the handshaking mechanism involves the data path gates (though the generation of the valid signal) it is not allowed to have route-through signals in a pipeline stage, since that would cause problems with the handshaking protocol and timing. Therefore this design style also requires the addition of buffers for all such signals.

2.5.4 STFB and SSTFB

Single-Track Full Buffer is a design style proposed in [18][7] for fine-grain pipeline design. It uses 1-of-N encoding for the data and also 2-phase single-track handshaking between gates that is embedded in the data. It has been shown to yield very high throughput designs. There are several features of this design style that contribute to its high performance capabilities. Firstly the gates use dynamic logic internally for higher performance and reduced area. Secondly the gates have extremely small forward latency of 2 transitions and a total cycle time of 6 transitions. That is accomplished by embedding the control signals as part of the data path and the use of 2-phase handshaking.

In STFB the sender will receive data and evaluate its output and then immediately tri-state its output. The receiver detects the presence of data and evaluates only when all the data has been received. This is done by properly designing the stacks of NMOS transistors so that all paths to ground use all inputs. When the receiver evaluates its outputs it will actively drive the wires low and then tri-state the inputs. This signals the sender that the data has been consumed and it can evaluate the next set of data. The data is encoded in a 1-of-N fashion therefore for each communication only one wire in the set will transition. This wire is therefore used simultaneously for the data, request and acknowledgment signaling between the two cells. Figure 8 shows an abstract view of such a communication channel as well as a timing diagram of two successive token transfers on a 1-of-4 channel of this type.

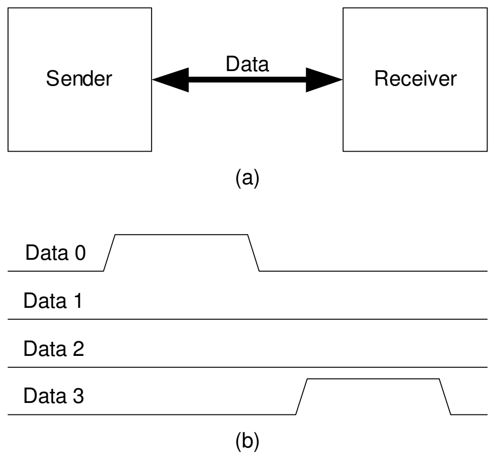


Figure 8 An abstract view of a STFB and SSTFB communication channel is shown in (a) and an example of two successive communications of a 1-of-4 such channel is shown in (b).

A problem with this template is that the data wires are not actively driven at all times. There are times that both transmitter and receiver will be in tri-state mode, hence the data becomes more susceptible to noise and leakage. Statisizers could be used to help alleviate this problem. An improvement for this protocol was recently published in [18] called the Static STFB, where the data is actively driven at all times. Here once the receiver detects the presence of data it actively holds the values present on the channel until it consumes it.

The high speed capabilities of this design style come at the expense of increased area, as expected from a fine-grain pipeline. The circuitry required on both ends of the channel, which is used for detecting, driving and resetting the data is not large, but substantial compared to the logic associated with the actual logic function of each gate. Another problem with this design styles is that since both ends of the channel actively drive the data lines at some time, the data communication has to be implemented with point-to-point communications, meaning that a gate cannot drive more than one more gate. Forks have to be implemented in special fork cells that are specifically designed for that purpose and which further increase the amount of logic required for implementation by adding gates that one would not usually need when using a different template.

2.6 Asynchronous Circuit Synthesis

2.6.1 Introduction

The lack of a comprehensive ASIC design flow for asynchronous circuits, like the one that exists for synchronous circuits, is probably the single most important reason for the limited adoption of asynchronous technology by the semiconductor industry. Several approaches have been proposed so far, but none of them has been able to address all the issues associated with asynchronous design. There are two major categories of design methodologies among the proposed work. One is attempting to define the problems according to an asynchronous specification that incorporates the exact description of all the operations and their relations. This approach is referred to as *High-Level Synthesis* and has the potential of creating circuits that are most optimally designed to match the given specifications. The other approach is to leverage off of tools that already exist in the market for synchronous flows and use those for the majority of the tasks associated with the design. That allows for faster turnaround times, since most tools already exist. The concept there is to synthesize synchronous netlists and then convert the results to an asynchronous design style.

2.6.2 High-Level Synthesis

Several approaches have been developed over the years for characterizing asynchronous circuits and then generating synthesized netlists based on the specification. These approaches defined languages able to handle the concepts necessary for the interpretations of asynchronous communications and the design process starts from specifications written in this language. Languages like CSP were defined to be able to define the problems in the form that is unique to asynchronous design. CSP defines a design in the form of concurrent hardware processes, and also the way that the processes communicate. Different variant have been defined, such as CHP that also have support for automated synthesis to a certain extent [28].

Phillips developed Tangram [21][7] which is a description language for asynchronous processes. Syntax based translation is used to convert the processes into circuits. The first step is to map the processes into some fixed handshake types of functional blocks that are predefined and classified according to their handshaking properties. These are called *handshake circuits* and are not mapped to any specific library. Then the netlist is broken up into combinational logic, registers and asynchronous elements and in the last step the combinational logic is taken into a commercial synthesis tool and synthesized using synchronous libraries.

Bardsley et. al proposed another language called Balsa and subsequent research has added onto this framework for top level synthesis [3][4][7]. This also starts from a new programming language that is made explicitly to understand asynchronous-specific commands and details. Balsa follows the synthesis approach of Tangram although there some differences in terms of the format and the features offered. Incremental synthesis is supported and the intermediate handshake circuits follow the original specification architecture. A new description format is used to describe the handshake circuits and the final mapping translates every part of the circuit into standard cell components of given libraries.

This approach in general is very promising, however it starts from a high level of abstraction that requires the use of a new language and also good understanding of asynchronous processes and interfaces. At the same time it requires more complete tool suites that will be used to bring the designs to the physical implementation level. Simulation and development tools are needed to assist with the design process, which are at a big disadvantage compared to their synchronous counterparts that have been in development for more than two decades. Even if that was not the case companies and engineers would be very reluctant to invest in new tools that have not been proven in the field already. Existing designs would also have to be recoded to be adapted to the new flow.

On the other hand there are several existing designs in HDL languages that are extremely useful and interesting and also a large number of designers that are able to code, partition and optimize designs using current synchronous flows. For asynchronous design to gain leverage in the market it is imperative that flows based on RTL level synthesis using conventional HDL and industry-accepted tools are available. In fact it is my belief that for a good designer that is used to a particular flow and its capabilities the results from the two approaches are going to be similar.

2.6.3 HDL-Based Synthesis

This type of synthesis starts from conventional HDL and uses tools that are well established in the industry for the implementation of synchronous circuits. Some of these approaches can use existing code that has already been coded for synchronous circuits and others require certain coding details that assist the translation process. Overall most of these approaches can be seen as synchronous translations, and especially for legacy circuits there is a possibility that they cannot take advantage of all benefits and features associated with asynchronous design. For example since timing information for the different processes is not available, trades between concurrency and sequencing, especially in terms of resource sharing are not possible unless the designer has already taken these into consideration when coding at the HDL level.

2.6.3.1 De-Synchronization

De-synchronization [8][11][13][14] is a method for directly translating synchronous netlists into asynchronous netlists with local handshaking between cells. Synthesis is performed from regular HDL with a regular synchronous flow. The netlist that is produced is then parsed and in the first step the flip-flops in the design are all replaced by pairs of latches. After that is done the clock circuitry is removed from the netlist and replaced by local controllers that implement local handshaking. Consequently, there is combinational logic that is followed by an odd pair of latches that is immediately followed by an even pair of latches with no logic in between. When the control is connected together delay matched lines have to be inserted in parallel with the combinational logic, namely between the even latch controllers and the odd latch controllers. The delay lines are used to match the worst-case delay of the combinational logic, and this is necessary since the data path is designed out of regular single-rail logic gates that come from the synchronous synthesis library.

The benefit of this approach is that commercially-available standard cell libraries and standard ASIC tools are used for the synthesis. Therefore no new libraries are necessary and the only tool that is required is the parser that does the flip-flop replacement and control instantiation. Another big benefit is that the circuit can easily be proven to be equivalent to its synchronous counterpart given that all timing constraints between the controllers are correctly verified. The correctness of the circuit is mathematically proven without any exhaustive analysis of the circuit. However, the final asynchronous circuit is so close to the originally synthesized netlist that no significant performance or area benefits are realized with this flow. The power associated with the clock distribution circuit is removed, which is a big benefit, but the circuit is constrained in terms of its capabilities from the synchronous specification.

2.6.3.2 Phased Logic

Phased Logic is another approach in the quest of automated HDL based synthesis, originally described in [22]. The flow was adapted and optimizations for both ASIC and FPGAs [40][34][35][36][37]. The flow begins with a synchronous HDL design. The design is synthesized using commercial tools and the netlist is then manipulated with custom tools to produce the final netlist. The gates are replaced by their equivalent counterparts that use two wires instead of one per signal. Each gate also has an internal state that it uses to handshake with neighboring cells.

The two wires are not used to generate dual-rail data, but instead are used to define not only the data, but also the state of the originating gate. This helps distinguish "odd" data from "even" data and the gates are now capable to distinguish when the data is valid and when not. When a gate is in a particular state – odd or even – it is ready to fire when all inputs have the same state as the gate. As soon as it fires the gate changes its state and waits for tokens with the same state value. It sends the output data with the original state of the gate that produced it so that the gates further down the pipeline can fire. Flip-flops are converted to buffers, but are initialized with a token upon reset. An innovation in this flow is that the circuit is analyzed to verify that the netlist initial conditions guarantee a netlist that is live and safe. This is guaranteed by making sure that all cycles have at least one token, by inserting buffers for flip-flop-to-flip-flop paths and making sure that each signal is part of a directed circuit that has one token. Dead gates are removed and if there are any liveness or safeness problems are resolved by adding additional logic. Other optimizations are available, such as slack-buffer insertion and use of units that can evaluate early, however parts of this flow are done manually with external files that include this information.

This is an integrated flow that has been used to generate circuits and proven its value. It has the benefit of an FPGA prototyping flow and it is continuously appended with tools that make it close to commercial standards (such as the PLFire schematic viewer [15]). However it requires custom gate design and also some inputs from the user for a successful conversion. For example slack-matching is done by manually specifying files that include the position and the number of buffers to be inserted.

2.6.3.3 Null-Convention Logic

Theseus Logic proposed Null Convention Logic (NCL) [15] for the implementation of asynchronous circuits. Later efforts have enhanced the flow and also proposed programmable solutions for implementing this kind of circuits [22][28][30]. Reconfigurable solutions have also been proposed for the particular design style and flow. NCL starts from conventional HDL, but the code has to be written strictly in RTL form (no behavioral register inference) and the register acknowledgment and request signals have to be specified. It then gets synthesized into an intermediate library called 3NCL. This library is still a single-rail library but with the addition of an extra possible value (the NULL value) for all wires. This preserves single-rail simulation and design capabilities, while emulating the final dual-rail gates. The final library is a full dual-rail library, but since the control is written around the fact that data will assume the value of NULL in a handshake cycle this is necessary for simulation. After this stage of verification a second run of synthesis is performed to translate the 3NCL gates into 2NCL gates that are the true dual-rail gates that will be used for the physical design process. In order to assure DI behavior only a limited variety of gates are used (2-input NAND, NOR, XOR)

Generally this is a complete flow, but still requires some input from the designer (register handshakes and placement). It uses existing synchronous tools for the majority of the design flow and achieves results that are close to manual designs. It also closely follows the original synchronous specification and does not optimize the design specifically to match the asynchronous library capabilities.

2.6.4 Challenges in RTL-Based Synthesis

A lot of different approaches have been proposed to achieve automated ASIC flows that start from HDL and result in finalized circuits using as much of the existing synchronous ASIC toolset as possible. All these flows focus on getting a design through the flow, but none address all the optimization issues that are specific to asynchronous design. Although useful, they fail to address issues that prevent the generation of circuits that can be superior to their synchronous counterparts.

The flows presented above depend on the synchronous netlist for the definition of the pipeline stages. This is not taking advantage of key features of asynchronous design styles and thus cannot yield more than incremental improvements. Moreover none of these methods have any good way of guaranteeing performance constraints, which is a basic requirement for any commercial design. Slack is not considered except for the case of Phased Logic, and even there this is done manually from externally defined input files that are user-specified.

A finer and more design-style aware pipelining mechanism can yield better results and this is the driving force behind our clustering approach. Tailoring the circuit pipeline to the performance requirements and the chosen design template improves the quality of results. With a framework for correctness and performance maintenance this leads to a more powerful method that has potential to yield results that are more competitive than the ones provided by existing flows.

2.7 General Design Considerations

2.7.1 Local and Global Cycle Time

In the absence of a global clock, asynchronous circuit performance is characterized using different metrics. When characterizing an asynchronous pipeline stage (could be as small as a single cell/gate for micro-pipelines) there are two important metrics to characterize performance. The first one is *forward latency (FL)* and is measured as the time between the arrival of a new token, when the pipeline stage is idle, and the production of valid outputs for the next stage. This is a metric that is only dependant on the internal design of the pipeline stage. The second metric is called the *local cycle time (LCT)*, and it is defined as the time between the arrival of a token and the time that the unit has reset itself back to the idle state and is ready to receive the next token. This number is generally affected by the next pipeline stages as well since the handshaking on the right side of the stage defines the time at which the stage can reset its output and proceed to get ready to accept new data. Both metrics are calculated during the design phase in terms of transitions, meaning the number of signal transitions that have to take place for the pipeline stage to move from one state to the next. Even though this is not directly translated into actual time, it is a useful first tool for tradeoff studies, design style comparison and performance estimation.

Once the local cycle time and forward latency is known there are several methods to do a more thorough analysis and find the performance of the entire circuit, and potentially identify the bottlenecks in the system. This is generally a very labor-intensive process that cannot be performed without a tool designed for this purpose, but the basic ideas can be intuitively described using the defined metrics of forward latency and local cycle time. The performance of a circuit is defined as the *global cycle time (GCT)* of the circuit and it is essentially the metric that defines how many transitions it takes the circuit to process a token on average. Ideally the global cycle time is equal to the maximum of the local cycle time and the *algorithmic cycle time (ACT)*. The algorithmic cycle time is the maximum for all cycles of the sum of the forward latencies of all the pipeline stages in the cycle divided by the number of tokens (data) that are in the cycle at any time. This is the maximum performance target for a design and the global cycle time cannot be improved beyond this point. However, the design might have a cycle time that is higher than this value, depending on the topology and the number of tokens in the design.

The reason that this might happen is that the performance is defined not only by how fast data can propagate down the pipeline, but how fast the pipeline resets to accept new tokens. The *backward latency (BL)* of a pipeline stage is defined as the difference between the local cycle time and the forward latency and it can be perceived as the time it takes for a *bubble* – or empty position in the pipeline – to propagate backwards in the pipeline. Alternatively, the backward latency can also be defined as the time it takes a node to complete the handshaking with its neighboring cells and reset itself so that the next token can go through.

The forward and backward latency combined define the performance of a local pipeline stage. However the alignment of the data in the forward direction as well as the alignment of the bubbles in the backward direction is important to guarantee that a given global cycle time is achievable even if both the ACT and LCTs are all smaller than the requested global cycle time. This concept of alignment between the handshakes of the various stages is called *Slack Matching* and due to its importance it will be discussed in further detail in Chapter 4.

2.7.2 Handling Forks and Joins

Now that there is a notion of performance defined for the circuit and the pipeline stages individually, it is easier to show what the problems are when defining an asynchronous pipeline. The first issue that every designer is faced with when designing an asynchronous circuit is dealing with forks and merges in the data path. Due to the fact that the handshaking signals required for synchronization propagate along with the data, extra gates usually have to be added to make sure that the request and/or acknowledgement signals get combined so that the correctness of the protocol is maintained. If care is not taken it is easy to have the system fail because of a deadlock at a join, due to improper acknowledgement (or request) along one of the two merging paths. These additional gates usually are on the critical path and have to be taken into consideration in order to avoid impeding performance. Also paths going through forks and joins are also likely to be parts of cycles or re-convergent paths and the cycle time analysis has to be performed to verify that the design performance is not reduced.

2.7.3 Fanout Optimization

Another aspect of forks is the handling of fanout. In synchronous design buffers can be added to high-fanout nets to improve performance without altering the basic functionality of the circuit. In asynchronous circuits, however there are cases where a high-fanout node has to be buffered, and the buffer that will be added alters the timing and structure of the circuit in such a way that affects the global cycle time. Even worse, in cases like SSTFB dedicated cells have to be inserted to handle nodes with fanout grater than one, and all these cells have to be included in the design in a way that does not cause performance degradation. This might imply modifying the shape of the fanout tree, or adding buffers in paths parallel to the one being altered. Generally this is a labor intensive process that currently needs to be undertaken manually during the design process.

2.7.4 Clustering

In design styles that are targeted towards coarse-pipelines such as MLD or even finer-grain pipelines such as MOUSETRAP, another problem arises that related to the placement of the gates within the different pipeline stages. The more logic one places in a stage the smaller the control overhead for the circuit. In the case of MLD for example, though, the wider the pipeline the slower the completion detection and therefore the circuit itself. The same is true for the number of logic levels within each stage. For MOUSETRAP the larger the grouping the slower the design, since the requests (and hence the cycle time of a pipeline stage) are dictated by the longest combinational path in the pipeline stage. Generally one can trade the clustering – or the distribution of gates within pipeline stages – on both dimensions, both in terms of depth and width, in order to achieve better performance. It is also obvious that the grouping can further affect performance, since one grouping could require less forks and joins to be inserted in the circuit than another. Generally this is another set of tradeoffs that are currently performed manually and it is up to a good designer to achieve a good distribution that boosts performance and reduces area.

Chapter 3 Clustering

Before going through the entire design flow and the tasks performed by both the commercial and customized pieces of software that comprise it, it is interesting to focus on the problem of clustering. This problem is common for all the different design styles, and it is equally important for all of them. The clustering of the gates inside larger pipeline stages allows the circuit to reduce the control overhead and make the different design styles competitive to not only each other, but also their synchronous counterparts.

The goal of this chapter is to formally define the problem and its solution and to lay the foundation for a successful application of clustering on any circuit. In particular, this chapter develops criteria that guarantees that clustering preserves the functionality of the circuit and does not introduce structures that make the circuit unable to meet its performance requirements.

3.1 Definitions

Circuits usually are designed subject to performance constraints that are derived from system requirements. Even though it is interesting to find the "fastest" a circuit can run or the "smallest" it can be made, practically it is not very useful, since the circuit requirements are always defined by system parameters that are not dictated by the circuit capabilities, but by the overall system function. Therefore we plan to define our problem in such a way that it can address a variety of design requirements.

The first step is to abstract the circuit into a more generic structure so that we can formulate our problem mathematically. This structure is a weighted directed graph $G=(V,E,h,m)$, where V is the set of nodes in the netlist. $V=PI\cup PO\cup CL\cup TB$, where PI is the set of primary inputs, PO is the set of primary outputs, CL is the set of combinational gates and TB is the set of flip-flops or TOKEN_BUFFERS. All four set PI,PO,CL,TB are mutually disjoint sets. E is the set of directed edges $E\subseteq(VxV)$. We will use the notation $e_{i,j}=(v_i,v_j)$ for an edge in E to simplify our notation for a directed edge that starts from node v_i and ends in node v_j. We also require that E does not contain any self-loops $e_{i,i}$. We also will define a function $h:E\rightarrow\Re^+$ that is used to map an edge onto a positive real number that represents the *forward latency* of the edge. We also define function $m:E\rightarrow\{0,1\}$, such that

$$m(e_{i,j})=\begin{cases}1, & v_i\in TB\\ 0, & otherwise\end{cases} \quad (1).$$

We define a path $p_{i,j}$ as a sequence of edges in E , the first edge in the sequence starting from node v_i, and the last edge in the sequence ending in node v_j and such that for all other edges in the sequence, their starting point is the ending point of the previous edge in the path and their ending point is the starting point of the next edge in the path. We also assume in this document that a path goes through each node once (simple path). We will also define a cycle as a path $p_{i,i}$ that starts and terminates at the same node v_i. We also define P_G as the set of all paths that exist in the G.

Another important input will be a target performance metric, which is defined in terms of the *target cycle time* (TCT) of the circuit and will be defined as τ_{goal}. We also define the *algorithmic cycle time* (ACT or $\tau_{a\lg}$) of the circuit, which is the lower bound of τ_{goal} beyond which τ_{goal} is no longer achievable. Thus $\tau_{goal} \geq \tau_{a\lg}$. Having defined a path the algorithmic cycle time is defined as:

$$\tau_{a\lg} = \max_{v_i \in V : \exists p_{i,i} \in P_G} \left\{ \frac{\sum_{e_{j,k} \in p_{i,i}} h(e_{j,k})}{\sum_{e_{j,k} \in p_{i,i}} m(e_{j,k})} \right\} \leq \tau_{goal} \text{ (2).}$$

We also define the weight of an edge $w_{i,j} = w(e_{i,j}) = h(e_{i,j}) - m(e_{i,j}) * \tau_{goal}$. We have the following convention:

$$w(e_{i,j}) = h(e_{i,j}) - m(e_{i,j}) * \tau_{goal} = \begin{cases} h(e_{i,j}) > 0, v_i \in PI \cup CL \\ h(e_{i,j}) - \tau_{goal} < 0, v_i \in TB \end{cases} \text{ (3).}$$

We also need to define the weight of a path – as an extension of the edge weight – that is the sum of the weights of all edges in the path sequence, so $w(p_{i,j}) = \sum_{e \in p_{i,j}} w(e_{i,j})$.We also define the length of a path as the number of edges in the sequence of the path. So $L(p_{i,j}) = |p_{i,j}|$.

We also define a distance between two nodes v_i and v_j as the maximum weight of all valid paths from v_i to v_j, or as $-\infty$ if no paths exist from v_i to v_j, which we denote as $d_{i,j} = \begin{cases} \max_{\forall p_{i,j} \in G} \{w(p_{i,j})\}, if \quad \exists p_{i,j} \in P_G \\ -\infty, otherwise \end{cases}$ (4).

We also define the *transitive fanout (TFO)* and *combinational transitive fanout (CTFO)* as well as *transitive fanin (TFI)* and *combinational transitive fanin (CTFI)* as follows:

$$TFO(v_i) = \{v_j : \exists p_{i,j} \in P_G\},$$

$$TFI(v_j) = \{v_i : \exists p_{i,j} \in P_G\}, \qquad (5)$$

$$CTFO(v_i) = \{v_j : \exists p_{i,j} \in P_G \wedge \neg \exists e_{k,l} \in p_{i,j} : v_k \in TB\}, \text{ and}$$

$$CTFI(v_j) = \{v_j : \exists p_{j,i} \in P_G \wedge \neg \exists e_{k,l} \in p_{j,i} : v_k \in TB\}.$$

So essentially the $CTFO(v_i)$ (and equivalently the $CTFI(v_i)$) is the set of all nodes that are reachable from v_i (or equivalently for *CTFI* that can reach node v_i) through a path that does not go through a TOKEN_BUFFER node.

We also have to formally define the *local move* operation before we begin our discussion of the clustering algorithm. A local move is a function on the graph $G = (V, E, h, m)$ that produces a new modified graph $G' = (V', E', h, m)$. It essentially takes two nodes $v_i, v_j \in V$ and replaces them in V' with a unified new node $v'_k \in V'$ that contains the contents of both nodes (in circuit terms that would be the instances and wires internal to the pipeline stages that correspond to the original nodes v_i and v_j). The rest of the nodes of V are preserved in V'. Mathematically:

$$V' = V - \{v_i, v_j\} + \{v'_k\} \quad (6)$$

If both $\exists e_{i,j} \wedge \exists e_{j,i} \in E$ the move is not allowed because this case would generate a self-loop in the graph (a cycle of length 1). Otherwise, when combining nodes v_i and v_j into the new node v'_k, the edges in the set E' are generated as follows:

- $\exists e_{m,i} \in E \vee \exists e_{m,j} \in E$ with $m \neq i, m \neq j$ then $\exists e_{m,k} \in E'$.
- $\exists e_{i,m} \in E \vee \exists e_{j,m} \in E$ with $m \neq i, m \neq j$ then $\exists e_{k,m} \in E'$. *(7)*
- $\exists e_{m,l} \in E$ with $m \neq i, m \neq j$ and $l \neq i, l \neq j$ then $\exists e_{m,l} \in E'$.

So the new node v'_k has the combined fanin and fanout of v_i and v_j, except for any edges between the two that get absorbed in the new node and are removed from the top-level graph. In other words, nodes $v_i, v_j \in V$ are replaced by a single node $v'_k \in V'$, and $v_i, v_j \in V$ are also replaced by $v'_k \in V'$ in all directed pairs (edges) in E'.

An important observation is that if either edge $e_{i,j}$ or $e_{j,i}$ exist in E a corresponding edge does not exist in E', which prevents the generation of a self-loop. So assuming that the initial netlist has no self-loops, no new ones can be created during the execution of local moves.

If an edge $e_{i,j} \in E$ gets absorbed, then $\forall e_{m,i} \in E$, $e_{m,k} \in E'$ it is true that $w(e_{m,i}) + w(e_{i,j}) \geq w'(e_{m,k})$. This means that an absorbed edge can increase the weights of all incoming edges to the new node $v'_k \in V'$ that before the execution of the move we incoming edges to v_i, but at most by $w(e_{i,j})$. If $e_{i,j} \notin E$ then the local move does not change the weights of any edges. It is also important to note that such a move is only allowed when $w(e_{i,j}) > 0$. This is due to the fact that $w(e_{i,j}) < 0$ implies from definition *(3)* that $v_i \in TB$. A move that absorbs a token buffer is not allowed because of the special functionality that token buffers serve in the circuit guaranteeing liveness around loops.

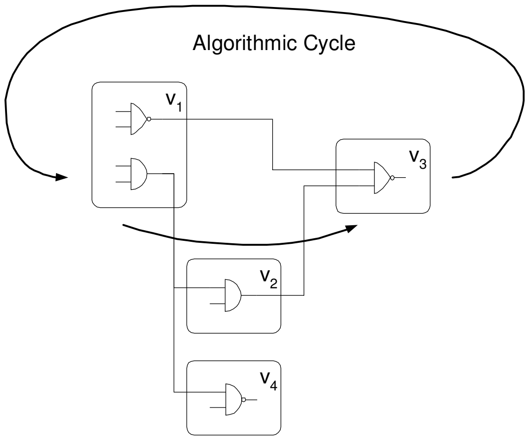
Finally it is important to note the following relationships, since they are very useful in understanding the effects of clustering on the connectivity of the graph model of the circuit. They represent the relationship between the *TFI, TFO, CTFI* and *CTFO* of the old nodes $v_i, v_j \in V$ and the new merged node $v' \in V'$ after the execution of a local move. These relationships can be easily derived from the definition of E' that was presented previously.

$$TFI(v') = \{TFI(v_i) \cup TFI(v_j)\} - \{v_i, v_j\}$$

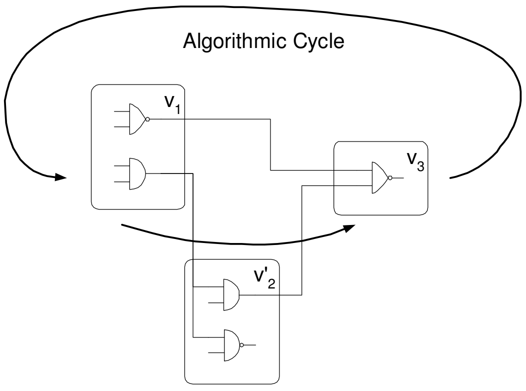
$$TFO(v') = \{TFO(v_i) \cup TFO(v_j)\} - \{v_i, v_j\} \qquad (8)$$

$$CTFI(v') = \{CTFI(v_i) \cup CTFI(v_j)\} - \{v_i, v_j\}$$

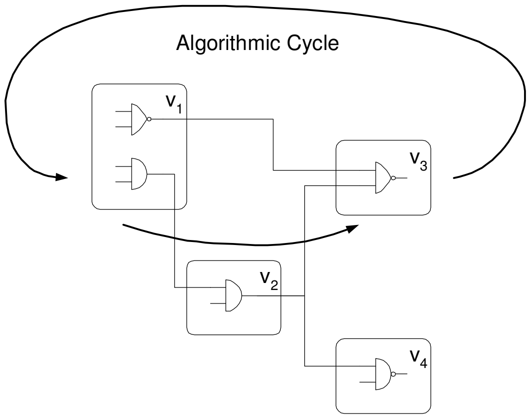
$$CTFO(v') = \{CTFO(v_i) \cup CTFO(v_j)\} - \{v_i, v_j\}$$


Algorithmic Cycle
v_1
v_3
v_2
v_4

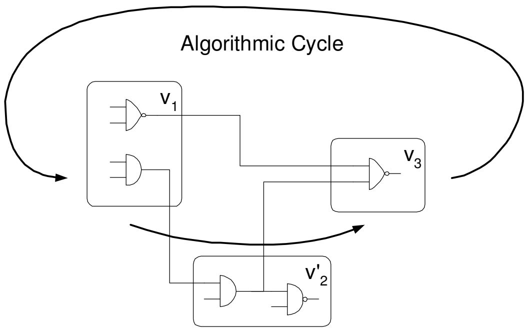
(a)


Algorithmic Cycle
v_1
v_3
v'_2

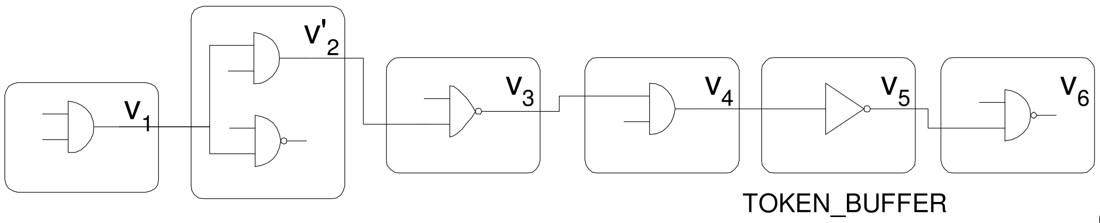
(b)


Algorithmic Cycle
v_1
v_3
v_2
v_4

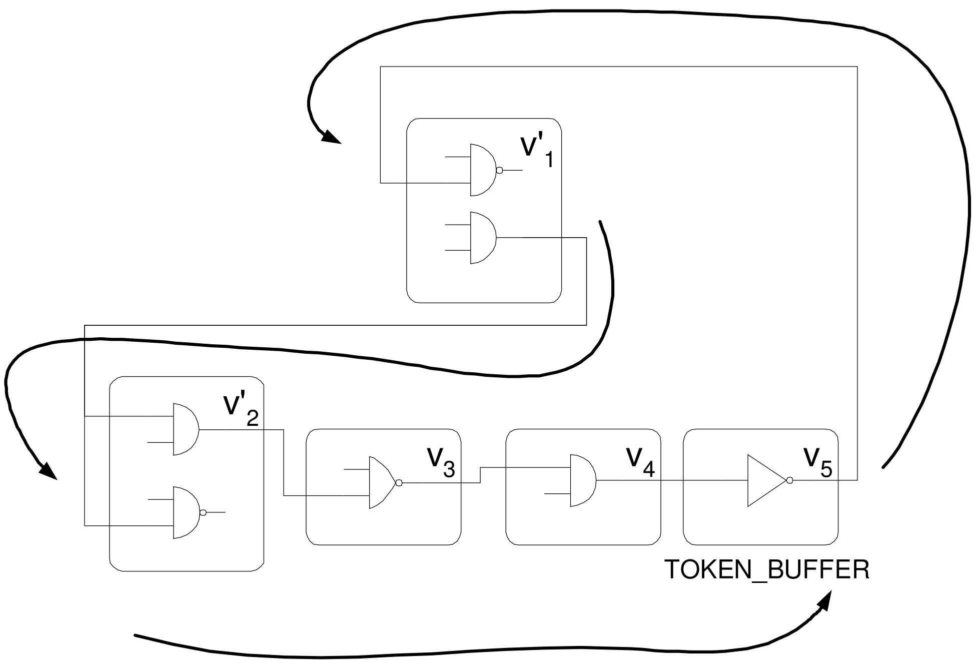
(c)


Algorithmic Cycle
v_1
v_3
v'_2

(d)


v_1
v'_2
v_3
v_4
v_5
v_6
TOKEN_BUFFER

(e)



(f)

Figure 9 Three different local move scenarios that have different impact on the $\tau_{a\lg}$ of the circuit. The circuit for the three scenarios before the move ((a), (c), (e) respectively) and after ((b), (d), (f) respectively) showing how the ACT could remain unaffected ((a),(b)), increase ((c),(d)), or even get introduced ((e),(f)).

Some examples of possible moves are shown in Figure 9. It is interesting to see the different scenarios and the effects that they could have either on the weight function $w(e_{i,j})$ and/or on the algorithmic cycle of the circuit:

- In the first scenario, which is depicted in Figure 9 (a) and (b) we can see an example of two nodes merging that are part of parallel paths and hence the execution of the move does not affect the ACT of the circuit, since the levels of logic in the nodes that are part of the path that defines the ACT are unaffected.

- In the second scenario, which is depicted in Figure 9 (c) and (d) there is an edge connecting the two nodes which gets absorbed. In this case the new logic will artificially inflate the ACT since a new level of logic is added in a node that is part of the critical path. Even though the levels do not really change in terms of the actual data path, this node will now have a delayed handshaking sequence due to the new logic, which affects the critical path. It should be noted here that additional merging with v_3 could remove this effect assuming this move was possible.

- In the third scenario, which is depicted in Figure 9 (e) and (f) there was no cycle in the portion of the circuit that is depicted. However after merging the two nodes v_1 and v_6 there is now a new cycle in the design. This cycle is again not introduced in the logic, since the circuitry is not modified by clustering, but introduced in terms of the control handshakes between the different nodes of the graph.

Having defined the general mathematical framework it is now time to look into a more formal representation of the clustering. In the following section we will define our present goals from the clustering and some key theorems that are necessary for the clustering to provide a functional solution that meets the user performance requirements.

3.2 Clustering Criteria

Clustering is a sequence of local moves that serve the purpose of minimizing control area. We have chosen to define local moves as the merging of two nodes, since all merges can be broken down into this basic two-way merge, and the two-way merge is easier to characterize and study. Since every pipeline cluster will ultimately need to have its own control unit as well as left and right C-element trees for multiple fanins and fanouts, every local move results in a drop in total area.

The ultimate goal is to find the clustering of the circuit into pipeline stages that achieves the minimum overall area while hitting a target performance. However practically this means that this has to take into consideration not only the clustering process, but also the effects of slack matching and fanout optimization. In general since this is a new area of research we found that before even considering different optimization algorithms and approaches, there were more fundamental problems that need to be addressed before area optimality. As we will show in Section 5.2.2 we chose a heuristic algorithm using a steepest descent approach and local constraints for the area optimizations. The focus of this work was maintaining correctness and performance during clustering. Having this foundation will allow us to further explore the optimization process in the future.

3.2.1 Ensuring liveness

The handshaking nature of asynchronous circuits requires that one constraint is satisfied to ensure the circuit is *live* [14] (also referred to as *liveness* of a circuit). Informally a circuit is *live*, if every cycle in the circuit should have at least one data TOKEN. This is guaranteed during the design process, by ensuring in every cycle in the design at least one TOKEN_BUFFER cell. A TOKEN_BUFFER is a special gate in the netlist that upon reset (or startup) will get initialized with a token (data). All other gates in the netlist are empty during initialization.

Based on the definitions in Section 3.1 the liveness criteria can be formalized in the context of our proposed graph model. A graph $G = (V, E, h, m)$ is live if every cycle $p_{i,i}$ includes at least an edge e that starts at a node $v \in TB$. Based on our convention, equivalently the graph is live if every cycle $p_{i,i}$ includes at least an edge $e : w(e) < 0$.

However, with arbitrary clustering it is easy to see how a cycle can get created that generates a new cycle that violates this principle. So our first task is to make sure that clustering does not destroy the liveness of a circuit and that we find a criterion that allows us to prevent all moves that could cause that from ever being executed.

Several of our proofs are based on the modified graph $G^* = (V, E - \{e_{i,j}, e_{j,i}\}, h, m)$, which does not include the two special edges that have the possibility of being absorbed during a local move. The reason for this is that those are the only two edges in the graph that are treated differently than others and by considering the modified graph that does not include them we can generalize our theory without having to consider the multitude of special cases. An important first conclusion is that in such a modified graph cannot include any paths between the two merging nodes that is of length 1. This is used in the proofs that follow.

Lemma 1: Let a local move merge two nodes $v_i, v_j \in V$ in graph $G = (V, E, h, m)$ into node $v'_k \in V'$ in the graph $G' = (V', E', h, m)$. In the modified graph $G^* = (V, E - \{e_{i,j}, e_{j,i}\}, h, m)$ if $\exists p_{l,n} \in P_{G^*}$ and $(l = i \vee l = j) \wedge (n = i \vee n = j)$ then $|p_{l,n}| > 1$.

Proof: Let us assume that $\exists p_{l,n} \in P_{G*}$. If $(l = i \vee l = j) \wedge (n = i \vee n = j)$ and $|p_{l,n}| = 1$. Then we will show that this statement cannot be true through contradiction. Indeed the possible cases are as follows:

1. $(l = n = i) \Rightarrow p_{i,i} = \{e_{i,i}\} \in P_G$ which contradicts our convention that $e_{i,i} \notin E$
2. $(l = n = j) \Rightarrow p_{j,j} = \{e_{j,j}\} \in P_G$ which contradicts our convention that $e_{j,j} \notin E$
3. $(l = i \wedge n = j) \Rightarrow p_{i,j} = \{e_{i,j}\} \in P_{G*}$ which contradicts our proposition that $e_{i,j} \notin E - \{e_{i,j}, e_{j,i}\}$
4. $(l = j \wedge n = i) \Rightarrow p_{j,i} = \{e_{j,i}\} \in P_{G*}$ which contradicts our proposition that $e_{j,i} \notin E - \{e_{i,j}, e_{j,i}\}$

Since all possible combinations contradict our original convention or proposition it is proven $|p_{l,n}| > 1$.▪

Now we would like to show that every path that exists in the modified original graph $G* = (V, E - \{e_{i,j}, e_{j,i}\}, h, m)$ also exists in the resulting graph after the move. This is an important conclusion since it helps us to easily qualify and evaluate the results of a possible move and its consequences in terms of the connectivity of the graph.

Lemma 2: Let a local move merge two nodes $v_i, v_j \in V$ in graph $G = (V, E, h, m)$ into node $v'_k \in V'$ in the graph $G' = (V', E', h, m)$. In the modified graph $G* = (V, E - \{e_{i,j}, e_{j,i}\}, h, m)$ if $\exists p_{l,n} \in P_{G*}$ then $\exists p_{o,q} \in P_{G'}$ such that

Case 1: If $(l \neq i \wedge l \neq j) \wedge (n \neq i \wedge n \neq j)$ then $o = l$ and $q = n$

Case 2: If $(l = i \wedge l = j) \wedge (n \neq i \wedge n \neq j)$ then $o = k$ and $q = n$

Case 3: If $(l \neq i \wedge l \neq j) \wedge (n = i \wedge n = j)$ then $o = l$ and $q = k$

Case 4: If $(l = i \wedge l = j) \wedge (n = i \wedge n = j)$ then $o = k$ and $q = k$

Proof: First we are going to prove all the cases for the special case that a path $p_{l,n} \in P_{G*}$ does not go though nodes $v_i, v_j \in V$, unless v_i, v_j are one of the endpoints of the path.

Case 1: $(l \neq i \wedge l \neq j) \wedge (n \neq i \wedge n \neq j)$ Assume that $\forall e = (v_1, v_2) \in p_{l,n}$ it is true that $v_1 \neq v_i$, $v_2 \neq v_i$, $v_1 \neq v_j$, $v_2 \neq v_j$, meaning the path does not include any edge that includes any of the merging nodes. Then based on the definition of the local move $\forall e_{1,2} \in p_{l,n}$ and $\exists e_{1,2} \in E'$ therefore $\exists p_{l,n} \in P_{G'}$.

Case 2: $(l = i \vee l = j) \wedge (n \neq i \vee n \neq j)$ $\forall e = (v_1, v_2) \in p_{l,n}$ $v_2 \neq v_i$, $v_2 \neq v_j$, then $\exists v_r \in V$ such that $p_{l,n} = \{e_{l,r}, p_{r,n}\}$, and from the definition of the local move if $\exists e_{l,r} \in E : l = i \vee l = j \quad \Rightarrow \quad \exists e_{k,r} \in E'$. So if $\exists p_{l,n} \in P_{G*}$ and $(l = i \vee l = j) \wedge (n \neq i \vee n \neq j)$ then $\exists p_{k,n} = \{e_{k,r}, p_{r,n}\} \in P_{G'}$.

Case 3: $(l \neq i \vee l \neq j) \wedge (n = i \vee n = j)$ and $\forall e = (v_1, v_2) \in p_{l,n}$ $v_1 \neq v_i$, $v_1 \neq v_j$, then $\exists v_r \in V$ such that $p_{l,n} = \{p_{l,r}, e_{r,n}\}$, and from the definition of the local move if $\exists e_{r,n} \in E : n = i \vee n = j \quad \Rightarrow \quad \exists e_{r,k} \in E'$. So if $\exists p_{l,n} \in P_{G*}$ and $(l \neq i \vee l \neq j) \wedge (n = i \vee n = j)$ then $\exists p_{l,k} = \{p_{l,r}, e_{r,k}\} \in P_{G'}$

Case 4: Now if we assume that $(l = i \vee l = j) \wedge (n = i \vee n = j)$ and the path does not include nodes v_i, v_j on any edge other than its starting and ending point. Using Lemma 1 $|p_{l,n}| > 1$, therefore we can write that $p_{l,n} = \{p_{l,r}, p_{r,n}\}$ for some node

v_r along the path. But if $\exists p_{l,r} \in G^* \Rightarrow \exists p_{k,r} \in P_{G'}$, and $\exists p_{r,n} \in G^* \Rightarrow \exists p_{r,k} \in P_{G'}$, so $\exists p_{k,k} = \{p_{k,r}, p_{r,k}\} \in P_{G'}$.

We have now proven all four cases for paths that do not go through nodes $v_i, v_j \in V$. We need to also show that all cases also stand when the path does go through nodes $v_i, v_j \in V$.

Next we cover the case where $\exists p_{l,n} \in P_{G^*}$ that goes through node $v_1 : (v_1 = v_i) \vee (v_1 = v_j)$ and $v_1 \neq v_l \wedge v_1 \neq v_m$. But then we can write that $p_{l,n} = \{p_{l,1}, p_{1,n}\}$. Using the conclusions of Cases 1-4 above on the sub-paths $p_{l,1}, p_{1,n}$ and for any value of l,n we know that if $\exists p_{l,1} \in P_G \Rightarrow \exists p_{o,k} \in P_{G'}$ and if $\exists p_{1,n} \in P_G \Rightarrow \exists p_{k,n} \in P_{G'}$. This implies $p_{l,n} = \{p_{l,k}, p_{k,n}\} \in P_{G'}$. So the relationships in Cases 1-4 stand also when the path goes through the nodes $v_i, v_j \in V$ ▪

With Lemma 2 we have shown that the new graph $G' = (V', E', h, m)$ after the local move maintains all the connectivity as that in the modified initial graph $G^* = (V, E - \{e_{i,j}, e_{j,i}\}, h, m)$. However, the initial graph $G = (V, E, h, m)$ is our real initial graph and therefore, it is useful to also show that the connectivity in G is maintained in G' even if the two special edges $e_{i,j}, e_{j,i}$ where part of some original path.

<u>Lemma 3:</u> Let a local move merge two nodes $v_i, v_j \in V$ in graph $G = (V, E, h, m)$ into node $v'_k \in V'$ in the graph $G' = (V', E', h, m)$. If $\exists p_{l,n} \in P_G$ and $p_{l,n} \neq \{e_{i,j}\}$, $p_{l,n} \neq \{e_{j,i}\}$ then $\exists p_{o,q} \in P_{G'}$ such that

<u>Case 1:</u> If $(l \neq i \wedge l \neq j) \wedge (n \neq i \wedge n \neq j)$ then $o = l$ and $q = n$

Case 2: If $(l = i \wedge l = j) \wedge (n \neq i \wedge n \neq j)$ then $o = k$ and $q = n$

Case 3: If $(l \neq i \wedge l \neq j) \wedge (n = i \wedge n = j)$ then $o = l$ and $q = k$

Case 4: If $(l = i \wedge l = j) \wedge (n = i \wedge n = j)$ then $o = k$ and $q = k$

Proof: Lemma 2 proves the proposition for $\forall p_{l,n} \in P_G : e_{i,j} \notin p_{l,n} \wedge e_{j,i} \notin p_{l,n}$. So we need to prove the proposition for $\forall p_{l,n} \in P_G : e_{i,j} \in p_{l,n} \vee e_{j,i} \in p_{l,n}$. Let us assume that $\exists p_{l,n} \in P_G : p_{l,n} = \{p_{l,i}, e_{i,j}, p_{j,n}\}$. Even though in $e_{i,j} \notin E'$ it is true that if $\exists p_{l,i} \in P_G$ then $\exists p_{l,k} \in P_{G'}$ and if $\exists p_{j,n} \in P_G$ then $\exists p_{k,n} \in P_{G'}$ therefore $\exists p_{l,n} \in P_{G'} : p_{l,n} = \{p_{l,k}, p_{k,n}\}$.▪

The three lemmas show that except for the single-edge path that connects the two merging nodes, the new graph G' includes all other paths that existed in G. This is a very useful conclusion that will be used to show the necessary conditions that need to be satisfied to maintain both liveness and performance on the new graph.

Theorem 1: Let a local move merge two nodes $v_i, v_j \in V$ in graph $G = (V, E, h, m)$ into node $v'_k \in V'$ in the graph $G' = (V', E', h, m)$. The graph G' is non-live iff in the modified graph $G^* = (V, E - \{e_{i,j}, e_{j,i}\}, h, m)$, $v_i \in CTFO(v_j)$ or $v_j \in CTFO(v_i)$.

Proof:

($\leftarrow$) Assume in G^* it is true that $v_j \in CTFO(v_i)$. Then from Lemma 1, $\exists p_{i,j} \in P_{G^*} : |p_{i,j}| > 1$. Consequently $\exists v_m \in CTFO(v_i) : v_m \neq v_j$ and $\exists p_{i,m}, p_{m,j}$ in G^* that only traverse combinational nodes. And from Lemma 2 it is also true that $\exists p_{k,m}, p_{m,k}$, so there exists a combinational cycle, therefore the graph G' is not live. Similarly we can show the same if we assume that $v_i \in CTFO(v_j)$, which proves the one side of the proposition.

($\rightarrow$) Now let us assume that G' is not live (but G was) and that $v'_k \in V'$ is part of a combinational cycle that was generated during the local move. We will assume that $v_i \notin CTFO(v_j)$ and $v_j \notin CTFO(v_i)$ and show that we reach a contradiction.

From the definition of the local move guarantees the absence of self-loops on nodes so $\exists v_m \in V'$ for which $\exists p_{k,m}, p_{m,k}$ in $P_{G'}$ and $v'_k \in CTFO(v_m)$ and $v_m \in CTFO(v'_k)$. But if $\exists p_{k,m}, p_{m,k}$ in $P_{G'}$ it is true due to Lemma 2 that $\exists p_{l,m}, p_{m,n}$ in P_{G^*} such that either:

- $l = i$ and $n = j \Rightarrow v_j \in CTFO(v_i)$, which contradicts our assumptions or
- $l = j$ and $n = i \Rightarrow v_i \in CTFO(v_j)$, which contradicts our assumptions or
- $l = n = i \Rightarrow G^*$ not live $\Rightarrow G$ not live, which contradicts our assumptions or
- $l = n = j \Rightarrow G^*$ not live $\Rightarrow G$ not live, which contradicts our assumptions.

Therefore all possible cases contradict our assumption, therefore if G' is not live, it must be that $v_i \in CTFO(v_j)$ or $v_j \in CTFO(v_i)$, therefore our proposition stands. ▪

Theorem 2: Let a local move merge two nodes $v_i, v_j \in V$ in graph $G = (V, E, h, m)$ into node $v'_k \in V'$ in the graph $G' = (V', E', h, m)$. Graph G' will be live if in the modified graph $G^* = (V, E - \{e_{i,j}, e_{j,i}\}, h, m)$, $d_{i,j} \le 0$ and $d_{j,i} \le 0$

Proof: From Theorem 1, we know that the new graph will not be live iff $v_j \in CTFO(v_i)$ or $v_i \in CTFO(v_j)$. But if $v_j \in CTFO(v_i) \Rightarrow \exists p_{i,j} : w(e) > 0 \forall e \in p_{i,j}$ by the definition of $CTFO$. And from the definition of $d_{i,j}$ it is clear that $d_{i,j} > 0$. So if $v_j \in CTFO(v_i) \Rightarrow d_{i,j} > 0$ and similarly if $v_i \in CTFO(v_j) \Rightarrow d_{j,i} > 0$. Therefore if $d_{i,j} \le 0$ and $d_{j,i} \le 0$, then $v_i \notin CTFO(v_j)$ and $v_j \notin CTFO(v_i)$, so the resulting graph will be live due to Theorem 1, which completes the proof. ▪

However it should be noted that the reverse argument is not always true, meaning that $d_{j,i} > 0$ does not guarantee that $v_i \in CTFO(v_j)$. For example, consider a situation of a long path from a PI through a TOKEN_BUFFER to a PO. It might be possible in that case to have a path for which $d_{i,j} > 0$, but which after the merge includes a TOKEN_BUFFER in the cycle. Such an example if presented in Figure 10.

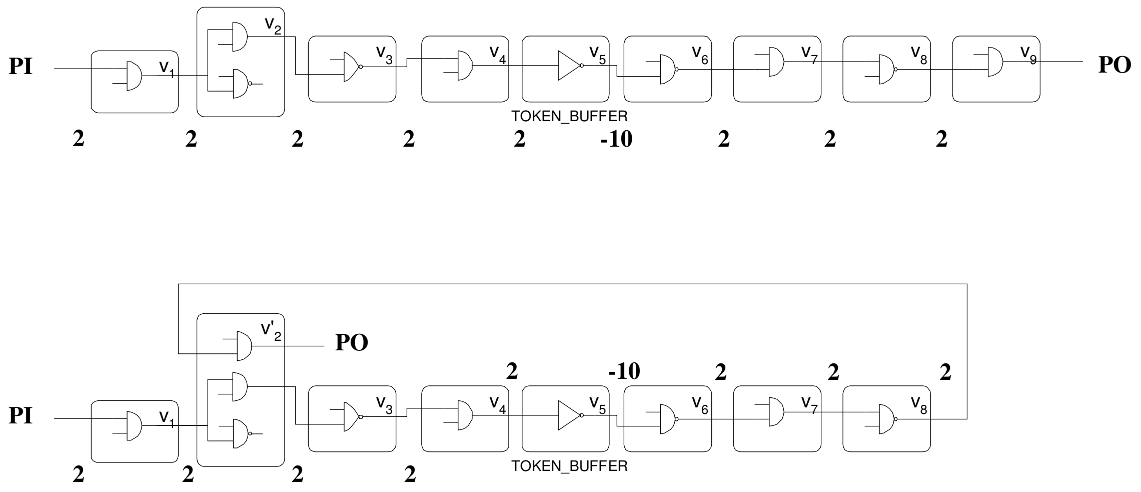


Figure 10 **An example where $d_{j,i} > 0$ does not guarantee that $v_i \in CTFO(v_j)$. Notice that $d_{2,9} = 2$, but the path includes v_5 is a TOKEN_BUFFER thus $v_9 \notin CTFO(v_2)$.**

Therefore this criterion is weaker than the one presented in Theorem 1, in the sense that it could exclude local moves that would not necessarily cause a non-live graph. The value of Theorem 2 is that once we have obtained the distances for the entire graph, the check can be implemented with a single lookup, while the operation of finding the *CTFO* is much more computationally intensive. It is also much more effective when combined with the performance criteria, which are discussed in the next section and are also based on distances. For example if in we like to avoid creating an algorithmic cycle that is longer than our target cycle time then this move should be avoided anyway. This will become more apparent in the following section, where the performance criteria are described.

3.2.2 Maintaining Performance

The goal of our analysis here is to define some criteria so that our local move operations can maintain the performance of the original circuit. There are two performance measures that are interesting here and one could potentially choose to even enforce them separately. The first one has to do with the TCT or τ_{goal}. In essence what that means is that the local move should not introduce any new cycles that could make the ACT any larger than the TCT thus making the circuit slower than requested by the user. The other one has to do with end to end latency. It is sometimes important that the PI-to-PO latency in a circuit does not increase. In that case we should be preventing any moves from being executed that could increase the latency from any PI to any PO in the circuit. Another limiting factor for the performance is the LCT. This is taken care of by local criteria that prevent moves from being executed that would slow down a channel to the point that it hurts performance. These are not discussed in this section but rather in the implementation section.

In order to prove our criteria we are first going to prove a Lemma that will help make our further discussion simpler. In particular we want to show that there are certain distance criteria that can be used to easily determine, whether the graph is satisfying the performance target for the TCT or not. This allows us to evaluate the performance of the graph (circuit) by marely evaluating the distances between the different nodes of the graph.

<u>Lemma 4:</u> In a live graph G the $\tau_{a\lg}$ is satisfied iff $\forall v_i \in V$ it is true that $d_{i,i} \le 0$.

Proof: For all nodes that are not part of any cycle it is trivially proven since in that case $d_{i,i} = -\infty$. So we need to prove this for all nodes that are part of a cycle. So let's assume that the graph G is live and also meets the $\tau_{a\lg}$. Then

$\tau_{goal} \geq \tau_{a\lg} = \max_{v_i \in V}\left\{\frac{\sum_{e_{j,k}\in p_{i,i}} h(e_{j,k})}{\sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})}\right\}$,or $\max_{v_i \in V}\left\{\frac{\sum_{e_{j,k}\in p_{i,i}} h(e_{j,k})}{\sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})}\right\} - \tau_{goal} \leq 0$, and since τ_{goal} is a positive constant this can also be transformed as:

$$\max_{v_i \in V}\left\{\frac{\sum_{e_{j,k}\in p_{i,i}} h(e_{j,k}) - \tau_{goal} * \sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})}{\sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})}\right\} \leq 0$$

We can further write that:

$$d_{i,i} = \max_{p_{i,i}\in G}\{w(p_{i,i})\} = \max_{v_i \in V}\left\{\sum_{e_{j,k}\in p_{i,i}}\left[h(e_{j,k}) - m(e_{j,k}) * \tau_{goal}\right]\right\}$$

$$= \max_{v_i \in V}\left\{\sum_{e_{j,k}\in p_{i,i}}\left[h(e_{j,k})\right] - \tau_{goal} * \sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})\right\}$$

And since $\sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})$ is the number of TOKEN_BUFFERS on that particular path and therefore and the graph is live $\sum_{e_{j,k}\in p_{i,i}} m(e_{j,k}) \geq 1$, so:

$$d_{i,i} \leq \max_{v_i \in V}\left\{\frac{\sum_{e_{j,k}\in p_{i,i}} h(e_{j,k}) - \tau_{goal} * \sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})}{\sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})}\right\} \leq 0$$

We can also trivially prove the reverse side of that relationship by contradiction. Assume that the graph is live, $\forall v_i \in V$ it is true that $d_{i,i} \leq 0$, but the $\tau_{a\lg}$ is violated, therefore

$\tau_{goal} \le \tau_{a\lg}$. Let us assume that the ACT is determined by a path $p_{i,i}$ for node $v_i \in V$. Then similarly to before we can write that:

$$d_{i,i} = w(p_{i,i}) = \sum_{e_{j,k}\in p_{i,i}} [h(e_{j,k}) - m(e_{j,k}) * \tau_{goal}] = \sum_{e_{j,k}\in p_{i,i}} [h(e_{j,k})] - \tau_{goal} * \sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})$$

But $d_{i,i} \le 0$ so

$$\sum_{e_{j,k}\in p_{i,i}} [h(e_{j,k})] - \tau_{goal} * \sum_{e_{j,k}\in p_{i,i}} m(e_{j,k}) \le 0 \Leftrightarrow \sum_{e_{j,k}\in p_{i,i}} [h(e_{j,k})] \le \tau_{goal} * \sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})$$

$$\Leftrightarrow \frac{\sum_{e_{j,k}\in p_{i,i}} [h(e_{j,k})]}{\sum_{e_{j,k}\in p_{i,i}} m(e_{j,k})} \le \tau_{goal}, \text{ since } \sum_{e_{j,k}\in p_{i,i}} m(e_{j,k}) \ge 1$$

$$\Leftrightarrow \tau_{a\lg} \le \tau_{goal}$$

Which contradicts the original assumption therefore the proposition stands. ▪

Lemma 5: Let a local move merge two nodes $v_i, v_j \in V$ in graph $G = (V, E, h, m)$ into node $v'_k \in V'$ in the graph $G' = (V', E', h, m)$. If G is live and the $\tau_{a\lg}$ and/or PI-to-PO latency constraints are satisfied and a local move is executed, any violating path in $G' = (V', E', h, m)$ will go through the newly formed node v'_k.

Proof: We can prove this through contradiction. Assume that we have a graph G where the $\tau_{a\lg}$ and/or PI-to-PO latency constraints are satisfied. Assume that the new node is $v_k \in V'$ that is the merge of $v_i, v_j \in V$. Assume that there exists a path that violates either the $\tau_{a\lg}$ or PI-to-PO latency constraints in $G*$ that does not go through the new node. However by the definition of the local move the edges into v_k are $\{e_{m,k} \in E'\} = \{e_{l,i} \in E\} \cup \{e_{n,i} \in E\} - \{e_{i,j}, e_{j,i}\}$ which means that if the path does not go through $v_k \in V'$ it cannot go through either $v_i, v_j \in V$. And since G and $G*$ are identical for all nodes and edges it means that if this violating path exists in $G*$ it existed in G as well. Therefore G must have a violating path, which contradicts our assumption. ▪

<u>Theorem 3:</u> Let a local move merge two nodes $v_i, v_j \in V$ in graph $G = (V, E, h, m)$ into node $v'_k \in V'$ in the graph $G' = (V', E', h, m)$. If G is live and the $\tau_{a\lg}$ constraints are satisfied, the local move will not create a path violating the $\tau_{a\lg}$ constraint iff in the modified graph $G* = (V, E - \{e_{i,j}, e_{j,i}\}, h, m)$ the following distance relationships are true $d_{i,j} + a \le 0$, $d_{j,j} + a \le 0$, $d_{j,i} + b \le 0$ and $d_{i,i} + b \le 0$ where $a = \begin{cases} 0, & if \neg \exists e_{j,i} \in E \\ w(e_{j,i}), & if \exists e_{j,i} \in E \end{cases}$

and $b = \begin{cases} 0, & if \neg \exists e_{i,j} \in E \\ w(e_{i,j}), & if \exists e_{i,j} \in E \end{cases}$

<u>Proof:</u>

<u>Case 1:</u> Let us first assume that $e_{i,j}.e_{j,i} \notin E$. Then based on the definition of the local move the weights of the edges of E are the same as those in E'. From Lemmas 3, 4 and 5 we can write the following relationship for the new distances d' in G' and the old ones d in G:

$$d'_{k,k} \le 0 \Leftrightarrow \begin{cases} d_{i,i} \le 0 & and \\ d_{j,j} \le 0 & and \\ d_{i,j} \le 0 & and \\ d_{j,i} \le 0 \end{cases}$$

In fact the first two relationships are true since $d_{i,i}$ and $d_{j,j}$ must be smaller or equal to 0 since G satisfies the $\tau_{a\lg}$.

<u>Case 2:</u> Let us first assume that $e_{i,j} \notin E \wedge e_{j,i} \in E$. Then $\exists e'_{l,k} \in p_{k,k}$ for which we know by definition that $w(e_{l,j}) + w(e_{j,i}) \geq w(e'_{l,k})$. This implies that the path that defines $d'_{k,k}$ could be greater by $w(e_{j,i})$ than the distance of the path it originated from if $e_{j,i}$ was not part of the original path of iit would otherwise be smaller or equal to it. This means that we can write:

$$d'_{k,k} \leq 0 \Leftrightarrow \begin{cases} d_{i,i} \leq 0 & and \\ d_{j,j} + w(e_{j,i}) \leq 0 & and \\ d_{i,j} + w(e_{j,i}) \leq 0 & and \\ d_{j,i} \leq 0 \end{cases}$$

The reason why the $w(e_{j,i})$ is not added to the other two paths is that since they end in node v_i it is impossible that they could be followed by $e_{j,i}$.

<u>Case 3:</u> Let us first assume that $e_{i,j} \in E \wedge e_{j,i} \notin E$. Then $\exists e'_{l,k} \in p_{k,k}$ for which we know by definition that $w(e_{l,i}) + w(e_{i,j}) \geq w(e'_{l,k})$. This implies that the path that defines $d'_{k,k}$ could be greater by $w(e_{i,j})$ than the distance of the path it originated from if $e_{i,j}$ was not part of the original path or it would otherwise be smaller or equal to it. This means that we can write:

$$d'_{k,k} \leq 0 \Leftrightarrow \begin{cases} d_{i,i} + w(e_{i,j}) \leq 0 \\ d_{j,j} \leq 0 \\ d_{i,j} \leq 0 \\ d_{j,i} + w(e_{i,j}) \leq 0 \end{cases}$$

The reason why the $w(e_{i,j})$ is not added to the other two paths is that since they end in node v_j it is impossible that they could be followed by $e_{i,j}$.

<u>Theorem 4:</u> Let a local move merge two nodes $v_i, v_j \in V$ in graph $G = (V, E, h, m)$ into node $v'_k \in V'$ in the graph $G' = (V', E', h, m)$. If G and G' are live and satisfy the τ_{goal} constraint, the move will not increase the latency of any path from PI to PO iff $\forall v_m \in PI$ and $\forall v_l \in PO$ it is true that:

$$d_{m,l} \geq \max\{d_{m,i}, d_{m,j}\} + \max\{d_{j,l}, d_{i,l}\}$$

<u>Proof:</u> The latency between $\forall v_m \in PI$ and $\forall v_l \in PO$ can be described by the distance between v_m and v_l that is $d_{m,l}$. The distance according to Lemma 5 will not increase if the length of all paths through the new node v'_k have length smaller or equal to $d_{m,l}$. We denote with d the distances of paths in P_G and with d' the paths in $P_{G'}$. The length of the longest path through v'_k can be written as $d'_{m,k} + d'_{k,l}$ therefore we can write that the statement will be true as long as $d_{m,l} \geq d'_{m,k} + d'_{k,l}$. For simplicity we are going to break down the proof into three cases:

<u>Case 1:</u> If $e_{i,j} \notin E \wedge e_{j,i} \notin E$. Assume that the local move generates a path from node v_m to node v_l. But since $e_{i,j} \notin E \wedge e_{j,i} \notin E$ based on the definition of the local move no edge will change weight so $d'_{m,k} = \max\{d_{m,i}, d_{m,j}\}$ and similarly $d'_{k,l} = \max\{d_{l,i}, d_{l,j}\}$. In other words the longest path from the input v_m to the new node v'_k is equal to the longest path to either of the two nodes that the move merges. Similarly the longest path from the new node v'_k to the output v_l is the longest path from either of the two nodes that the move merges. So the longest path from input v_m to output v_l that goes through

the new node v'_k will have a length of $d'_{m,k} + d'_{k,l} = \max\{d_{m,i}, d_{m,j}\} + \max\{d_{j,l}, d_{i,l}\}$. In order for the latency not to increase from input v_m to output v_l we can therefore write:

$$d_{m,l} = d'_{m,l} \geq d'_{m,k} + d'_{k,l} = \max\{d_{m,i}, d_{m,j}\} + \max\{d_{j,l}, d_{i,l}\}$$

Case 2: If $e_{i,j} \in E \wedge e_{j,i} \notin E$. In this case we know by definition that $w(e_{m,i}) + w(e_{i,j}) \geq w'(e_{m,k})$ and $w(e_{i,j}) > 0$, so a path in the original graph from input v_m to node v_i could get longer by $w(e_{i,j})$. However it is true that $d_{m,j} \geq d_{m,i} + d_{i,j} \geq d_{m,i} + w(e_{i,j})$. So the equation proven in Case 1 still holds, in fact we can simplify it and write $d_{m,l} = d'_{m,l} \geq d'_{m,k} + d'_{k,l} = d_{m,j} + \max\{d_{j,l}, d_{i,l}\}$

Case 3: If $e_{i,j} \notin E \wedge e_{j,i} \in E$. Similarly due to symmetry with Case 2 the equation that was proven in Case 1 holds and can be simplified and written as $d_{m,l} = d'_{m,l} \geq d'_{m,k} + d'_{k,l} = d_{m,i} + \max\{d_{j,l}, d_{i,l}\}$.

3.2.3 Modified Floyd-Warshall for finding distances

The Floyd-Warshall algorithm is used to find all pair-wise distances in the graph so that the constraints set above can be checked quickly by a simple look-up. The algorithm is originally designed to find all the minimum pair-wise distances in a graph and cannot be used if there are any negative-weight cycles in the graph. This works well for our case, because we need to find the maximum distances between all pairs of nodes. And assuming that the $\tau_{a\lg}$ is met in the graph originally according to Lemma 4 $\forall v_i \in V$ it is true that $d_{i,i} \leq 0$. Therefore there are no positive weight cycles in the graph, which allows to replace all min operations in the original algorithm with max operations, and still achieve convergence.

The complexity of the algorithm is $\Theta\left(|V|^3\right)$, so it is very expensive computationally. Moreover, the complexity does not change no matter how many nodes really need to be updated. After each local move is executed, the pairwise distances in the graph change and an update is required. Running the entire Floyd-Warshall algorithm was attempted, but it was quickly realized that this was impractical and so slow that made the use of the distance-based algorithms impractical.

A local update has been implemented that updates the array with simple operations only around the neighborhood of the new node after each move so that the Floyd-Warshall algorithm needs to run only once. However, the complexity of even one execution of the algorithm is really prohibitive for circuits that include tens of thousands of nodes. This is one of the reasons why when it comes to liveness two theorems were developed. Theorem 1 only requires a local search and in very large netlists where calculating all the distances is hard to do, one can choose to ignore performance in order to obtain results quickly and still maintain a functional circuit in the end of the operation, which however may or may not meet the performance requirements.

The local update that was developed to speed up the processing between move executions is extremely fast, reducing the overhead of the Floyd-Warshall to practically just the initial run that finds the initial distances. It takes advantage of the knowledge of the graph interconnect as well as the nature of the move so that it can speed up the processing and avoid updating any unnecessary values. The algorithm is shown in Figure 11.

The update function for the distances (update_distance_from_node) has complexity $O(B|N|)$, where *B* is the average branching factor at each node and $|N|$ is the number of nodes in the graph. It is based on the realization that each path from a node has to go through one of its fanout nodes, therefore all the distances from a given node can be calculated using just the distances to its fanout nodes and the distance vectors stored at each of the fanout nodes. This version of the algorithm was used extensively in our implementation of the distance algorithm presented in 3.2.2 and in practice it performs several times better than the Floyd-Warshall algorithm, however theoretically its performance could not be proven to be any better than the $O(|N|^3)$ that the Floyd-Warshall algorithm achieves, since this algorithm will not prevent a node from being visited several times.

```
Initialize
{
        Run Floyd Warshall algorithm
}

Update(newNode){
    Reinitialize_distance(newNode)
    Add newNode to searchlist

    While searchlist not empty {
       remove node from searchlist;
       update_distances_from_node(node);
       if any distance changed
          for each $s \in FI(node)$ not already in the searchlist
             add s to searchlist;
    }
}
Reinitialize_distance(s) {
    for all n
       if $n \in FO(s)$
          $d(s,n) = w(e_{s,n})$;
       else
          $d(s,n) = -\infty$;
}
update_distances_from_node(s) {
    for all $n \in FO(s)$
       for all $m \in G$
          $d(s, m) = \max\{d(s, m), d(s, n) + d(n, m)\}$;
}
```

Figure 11 Pseudo-code of the first distance-update routine. This version was found to be very fast in practice, but its performance is not bound.

A slightly modified version was then generated that has a performance that can be bound by $\mathrm{O}\left(B|N|^2\right)$, where B is the average branching factor at each node and $|N|$ is the number of nodes in the graph. The modified version is shown in Figure 12. This algorithm takes advantage of the fact that the pair wise distances in the rest of the graph did not change. So first the distances of the new node to all other nodes are reset and an update is executed to calculate the distances from the new node to all other nodes using just information from its fanout. Any path from the new node to any other node has to go through its fanout so this operation is enough to give us the new distances. Then the update is executed on the fanins of the new node, so that any new paths generated by the new node are updated on its fanin. Since the distance from all other nodes to the fanins of the new node did not change a final update on all nodes of the graph using these three nodes is enough to update the entire graph. This last loop is the longest operation in this update algorithm and is executed $\mathrm{O}(B|N|)$ times each containing $|N|$ updates so the total complexity of the update proposed is $\mathrm{O}\left(B|N|^2\right)$. For the typical circuits that we have studied it is true that the branching factor B is negligible in size compared to the number of nodes $|N|$ in the graph and thus for such graphs we can say that the complexity of the update is of complexity approximately equal to $\mathrm{O}\left(|V|^2\right)$.

This algorithm was designed, but not implemented. The previous version was proven to be very fast in practice and the results were satisfactory, so this new algorithm was not deemed necessary. Thus no run-time comparisons were generated to compare the two.

```
Initialize
{
        Run Floyd Warshall algorithm
}

Update (newNode) {
   Reinitialize_distance (newNode);
   update_distances_from_node (newNode);
   for all m $\in$ FI(newNode)
      update_distances_from_node_through_node (m, newNode);
   for all m $\in$ G
      for all n $\in$ FI(newNode)
         update_distances_from_node_through_node (m, n);
}
Reinitialize_distance(s) {
   for all n
      if n $\in$ FO(s)
         d(s, n) = w($e_{s,n}$);
      else
         d(s, n) = - $\infty$;
}
update_distances_from_node(s) {
   for all n $\in$ FO(s)
      for all m $\in$ G
         d(s, m) = max {d(s, m), d(s, n) + d(n, m)};
}
update_distances_from_node_through_node (s, p) {
   for all m $\in$ G
       d(s, m) = max {d(s, m), d(s, p) + d(p, m)};
}
```

Figure 12 Pseudo-code of the second distance-update routine. This version can be shown to have performance $O(BN^2)$, where B is the average branching factor at each node and N is the number of nodes in the graph.

3.3 Experimental Results

Both aspects of the clustering theory were tested using our software platform and ASIC flow that will be discussed in Chapter 5. In all cases we use the same clustering core algorithm, which is a greedy, steepest descent algorithm that picks the local moves that reduce the total area of the design the most. It stops when the algorithm determines that there are no possible moves left that do not violate the rules defined in each case. Since our slack matching and clustering rules also depend on the assumption that all LCTs are smaller all equal to the $\tau_{a\lg}$ and hence not critical, another local check was added that ensures that no moves are permitted that would grow an LCT beyond the allowed TCT target. Depending on the design style selected, other minor local constraints are also enforced to guarantee legal circuit implementations for the particular design style, but these details are minor and are not going to be discussed here.

We use for testing various examples from the ISCAS benchmark set that includes examples that are purely combinational (hence include no cycles) or mixed sequential and combinational (include state elements and cycles). We also added a few examples that are generally common in commercial circuit designs. We use four variants of clustering for each of the netlists. The first one applies no rules to the greedy clustering. The second one only applies the liveness rule described in Theorem 1. The third one adds the TCT constraints of Theorem 3 and also uses the expressions in Theorem 2 for liveness to reduce the computational complexity. Finally the last one adds the criteria of Theorem 4 to control the PI-to-PO overall latency of the circuit. For the examples we report the logic area after clustering, the total circuit area after slack matching and the GCT as measured by our software after clustering and slack matching is complete.

	Local & Global Cycle Time (transitions)							
	Area Guided		Liveness		Liveness & TCT		Liveness, TCT & Latency	
Design	LCT	ACT	LCT	ACT	LCT	ACT	LCT	ACT
c3540	28	DL	32	32	34	34	30	30
MAC16	28	68	32	32	34	34	28	28
MAC32	32	72	32	32	32	32	32	32
s1196	34	34	34	34	32	32	30	30
s1238	28	DL	32	32	34	34	30	30
s13207	28	DL	32	32	30	30	30	30
s1423	28	DL	32	50	32	32	32	32
s1488	30	30	30	30	32	32	28	28
s15850	30	DL	38	64	34	34	34	34
s27	26	26	26	26	24	26	24	26
s298	26	26	26	26	26	26	26	26
s344	26	26	26	26	28	28	26	26
s349	26	26	26	26	26	26	26	26
s382	26	DL	26	26	26	26	26	26
s386	28	28	28	28	26	26	26	26
s400	26	DL	26	26	26	26	26	26
s420	26	26	26	26	26	26	26	26
s444	26	DL	30	30	26	26	26	26
s510	28	28	28	28	28	28	28	28
s526	26	DL	32	32	28	28	30	30
s5378	32	32	32	32	32	32	30	30
s641	26	DL	32	32	28	28	26	26
s713	26	DL	32	32	32	32	26	26
s820	28	DL	32	32	30	30	28	28
s832	32	32	32	32	32	32	28	28
s838	26	DL	32	32	30	30	26	26
s9234	32	32	32	32	32	32	26	26
s953	28	DL	32	32	32	32	26	26
SISO	34	DL	36	76	46	46	30	34

Table 1 Performance of the different circuits in terms of Local and Algorithmic Cycle Time under different local move approval criteria. It is worth noting that without the use of our proposed criteria more than half of the circuits resulted in deadlock (marked DL).

Table 1 shows that 15 of 29 designs (51.7%) resulted in deadlock and were not functional after clustering. So it is clear that it is very important to be able to maintain the liveness of the circuit throughout these transformations, since the likelihood of destroying its functionality is very high. It is also important to note that a non-live circuit will also fail slack matching due to infeasible constraints, therefore in most cases the area guided results are skewed since there were no slack buffers added. Since such results are not meaningful area comparisons for the area-guided approach are not further analyzed.

	Logic Area (μm^2)			Total Area (μm^2)		
Design	Liveness	Liveness & TCT	Liveness, TCT & Latency	Liveness	Liveness & TCT	Liveness, TCT & Latency
c3540	53906.7	55238.4	67044.1	95369.5	92305.2	89650.9
MAC16	73659	79578	123170	218770	220988	209570
MAC32	61990.7	70978.2	84036.9	111006	140453	118325
s1196	22883.9	24017.5	27561	41813.6	45172.8	42873.4
s1238	24158.6	24916.1	28344.4	51539.3	54246	44670.5
s13207	132051	134391	144565	265536	273949	254701
s1423	26546.64	28230.9	31879.8	33606.1	59998.5	58762.9
s1488	29227.4	30055.7	31940.4	80592.8	80444.2	71767.3
s15850	117041.6	182966	182966	150210	297931	297931
s27	387.072	728.066	705.026	1013.76	1041.41	1018.37
s298	5170.18	5276.16	5234.69	8787.46	8865.79	8603.14
s344	5852.18	6165.54	6690.8	11796.5	12192.8	11699.7
s349	5824.5	6128.6	6667.8	12215.8	11828.7	11621.4
s382	6981.09	7326.7	7787.53	12575.2	13261.8	12722.7
s386	6420.66	6494.43	6570.98	11858.1	11259.1	12630.5
s400	7068.65	7455.76	7847.39	11994.6	12432.4	11713.5
s420	7225.34	7414.25	8262.14	15091.2	15390.7	14114.3
s444	6469.61	6907.43	7828.96	11649	12073	11593.7
s510	11920.9	12354	12570.7	31085.6	30034.9	27795.5
s526	8068.6	8193	9202.22	16132.6	16372.2	15699.5
s5378	61099.2	63975.6	68370.9	127436	129294	121197
s641	7326.73	7294.5	9400.3	15777.8	14667.3	14008.3
s713	7308.3	7345.12	9321.96	17459.7	14418.4	14008.3
s820	14742.8	15249.6	15931.6	33202.4	33225.4	29654.2
s832	13916.2	14303.3	15727.1	28781.6	30795.3	26993.7
s838	15694.9	15819.3	16985.1	34260.5	36159	29205.5
s9234	46361.1	48940.4	51255.4	79276	95743.9	91948.6
s953	18312.2	18266.1	20312.09	33039.4	34956.3	29707.8
SISO	22939.8	43118	100449.6	22939.8	167119	114762

Table 2 Area summary for different examples using different local move approval criteria. The left columns correspond to the logic area, which is the area of the clustered gates before the introduction of slack matching. The columns on the right correspond to the total circuit area after slack matching.

	% Logic Area Increase			% Total Area Increase		
Design	Liveness	Liveness & TCT	Liveness, TCT & Latency	Liveness	Liveness & TCT	Liveness, TCT & Latency
c3540	0.0%	2.5%	24.4%	0.0%	-3.2%	-6.0%
MAC16	0.0%	8.0%	67.2%	0.0%	1.0%	-4.2%
MAC32	0.0%	14.5%	35.6%	0.0%	26.5%	6.6%
s1196	0.0%	5.0%	20.4%	0.0%	8.0%	2.5%
s1238	0.0%	3.1%	17.3%	0.0%	5.3%	-13.3%
s13207	0.0%	1.8%	9.5%	0.0%	3.2%	-4.1%
s1423	0.0%	6.3%	20.1%	0.0%	78.5%	74.9%
s1488	0.0%	2.8%	9.3%	0.0%	-0.2%	-11.0%
s15850	0.0%	56.3%	56.3%	0.0%	98.3%	98.3%
s27	0.0%	88.1%	82.1%	0.0%	2.7%	0.5%
s298	0.0%	2.0%	1.2%	0.0%	0.9%	-2.1%
s344	0.0%	5.4%	14.3%	0.0%	3.4%	-0.8%
s349	0.0%	5.2%	14.5%	0.0%	-3.2%	-4.9%
s382	0.0%	5.0%	11.6%	0.0%	5.5%	1.2%
s386	0.0%	1.1%	2.3%	0.0%	-5.1%	6.5%
s400	0.0%	5.5%	11.0%	0.0%	3.6%	-2.3%
s420	0.0%	2.6%	14.3%	0.0%	2.0%	-6.5%
s444	0.0%	6.8%	21.0%	0.0%	3.6%	-0.5%
s510	0.0%	3.6%	5.5%	0.0%	-3.4%	-10.6%
s526	0.0%	1.5%	14.0%	0.0%	1.5%	-2.7%
s5378	0.0%	4.7%	11.9%	0.0%	1.5%	-4.9%
s641	0.0%	-0.4%	28.3%	0.0%	-7.0%	-11.2%
s713	0.0%	0.5%	27.6%	0.0%	-17.4%	-19.8%
s820	0.0%	3.4%	8.1%	0.0%	0.1%	-10.7%
s832	0.0%	2.8%	13.0%	0.0%	7.0%	-6.2%
s838	0.0%	0.8%	8.2%	0.0%	5.5%	-14.8%
s9234	0.0%	5.6%	10.6%	0.0%	20.8%	16.0%
s953	0.0%	-0.3%	10.9%	0.0%	5.8%	-10.1%
SISO	0.0%	88.0%	337.9%	0.0%	628.5%	400.3%
Average	**0.0%**	**11.5%**	**31.3%**	**0.0%**	**30.1%**	**15.9%**
Average excluding SISO,s1423,s15850				**0.0%**	**2.6%**	**-4.4%**

Table 3 Area increase for different examples using different local move approval criteria On the columns on the left we see the area increase for the clustered netlist before the addition of slack matching. On the columns of the right we see the area increase for the final netlist after slack matching,

From the Table 2 and Table 3 we can see that the simpler *CTFO* criterion-based algorithm performs the most amount of clustering, as expected, due to fewer constraints. The logic area for this algorithm is the smallest of the three variants analyzed. The addition of the ACT maintenance criteria poses extra constraints, yielding clustered netlists that have 11.5% more area. There is a total area increase on average of 30.1% once the slack matching is factored in. However if we exclude the three examples where CTFO yielded extremely large GCT essentially requiring no slack matching (SISO, s15850 and s1423) the average increase in total area is only 2.6%. Adding the latency constraints further constrains clustering, and one can see that the clustered area increase is now 31.3%. The slack matching cost drops though, since the latency constraints were intended to force better alignment of paths and reduce slack matching as well as maintaining latency. On average after slack-matching, this algorithm suffers a 15.9% area increase. In fact, if we again exclude the 3 examples that the liveness-only algorithm yielded excessive GCT on, this third algorithm is actually 4.4% better on average than the liveness-only one on total area. If one also factors in the fact that the latency-aware algorithm beat the others almost uniformly on the final GCT it is clear that it is the best one of the three.

Intuitively this is expected as this additional criterion is likely to force better alignment of nearby pipeline stages and prevent merging of stages that are further apart from each other. In general given that this algorithm is able to maintain performance and liveness. However it should be pointed out that this algorithm requires all pairwise distances in the circuit, which is a computationally expensive operation. On the other hand the liveness-only one has small complexity and thus may be more practical for larger circuits.

Chapter 4 Slack Matching and Fanout Considerations

4.1 Introduction

This section will discuss the concept of slack matching and improvements that have been made to existing algorithms. Although describing the entire theory behind slack matching as well as its modeling and formulation is beyond the scope of this work, it is useful to show the basic principle behind its use and motivation through an example. This allows the reader to understand the general context of the optimizations described here, without getting into some of the more intricate details of the process.

Slack matching can be thought as the process of properly aligning the timing of the handshakes between the pipeline stages in the design, so that a circuit can maximize its performance. If a stage generates data late, forcing another stage to wait then the receiving stage will have to stall its other inputs as it cannot process without all its inputs being present. This forward latency matching is straightforward and analogous to the latency matching that is frequently performed in synchronous circuits. In asynchronous circuits, though, it is also true that there is a backward latency, that defines the time it takes a stage to reset itself and get ready to receive data again. This forces an alignment constraint for the backward latencies as well, which is harder to visualize and which is realized in re-convergent paths. In both cases the mismatches can be handled by adding additional pipeline stages called *slack buffers*. The slack cells commonly are faster than regular logic cells and that allows them to be able to address larger mismatches than a common cell due to their excess *slack* which is defined that the difference between the circuit GCT and the cell LCT.

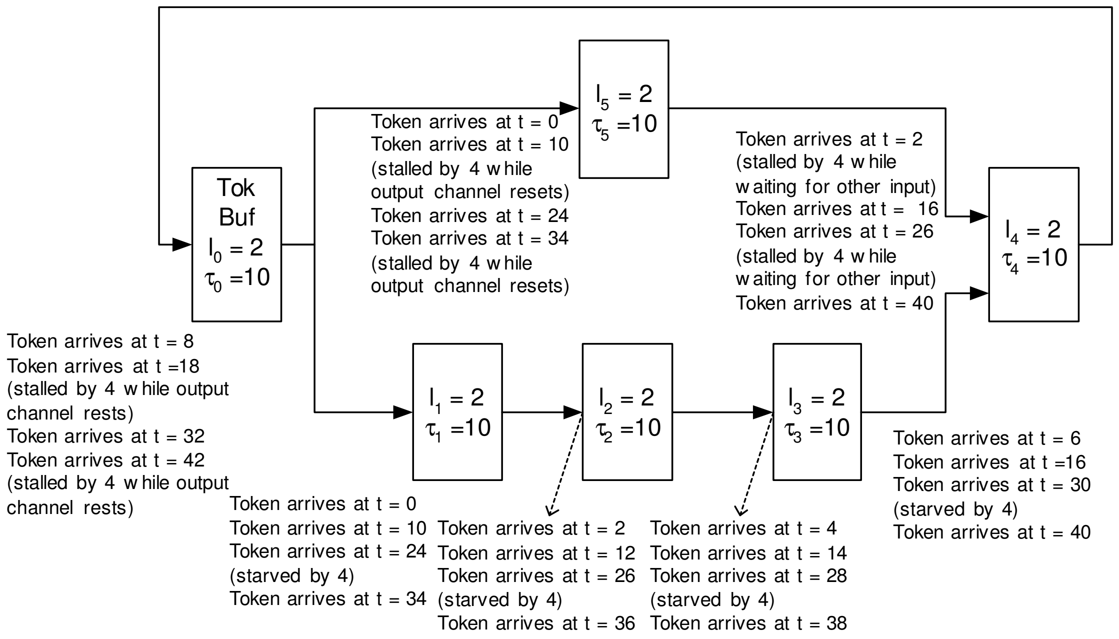


Figure 13 This example illustrates the effects of improper slack matching in a pipeline. The circuit shown above comprises pipeline stages that can run at a 10-transition cycle, however due to the mismatch the circuit cannot run faster than a 12 transition cycle.

The above intuitive explanation is illustrated in Figure 13. Notice that all pipeline stages can run at a cycle time of 10- transition delays. This should enable the circuit to also run at that speed, but due to improper slack matching it can not go faster than an average cycle of 12 transitions. One can start tracing the causes of this slow down by looking at the arrival times of the data at the different stages assuming that data is launched at time t = 0. Notice that the token from the top branch arrives at stage 4 at time t = 2. However the other branch is not ready causing it to wait for 4 transitions. This is the first stall and is happening due to the top path being too fast. Now stage 5 can only start resetting at time t = 6 when the data is used by stage 4, so it cannot accept data until t = 14. This in turn causes a stall between stages 0 and 5. The stalls inevitably keep propagating backwards, and on average it turns out that the circuit can never exceed 12 transitions per cycle performance on average.

The slack matching problem is commonly modeled using a Petri net model commonly referred to as the Full-Buffer Channel Net model proposed in [5]. As the name suggests the underlying assumption is that the stages are full-buffer stages, although in practice it has been shown to work well even for half-buffer cells. The problem can be presented as MILP problem, but due to complexity it has been also approximated in [5] with an LP. This section is devoted on optimizations that can be performed on this original formulation.

4.2 Slack Matching Fanout Tree Improvements

Slack matching adds buffers on the connections between pipeline stages to enhance the performance of the circuit. In reality slack matching creates a small buffer tree at the output of a cluster that ensures the alignment of the data at the leaf cells of the tree at the desired times. The cost of slack matching is high in many asynchronous circuit templates that could account to up to 33% of the total circuit area. Therefore it is worth investigating improvements to the existing models and methods, which could be used to reduce the area overhead that makes this process so costly.

Slack matching is done using the Full-Buffer-Channel-Net (FBCN) model as presented in [5]. In the FBCN model slack matching is done on channels. Channels are point-to-point edges in the graph and are used to describe abstractly a connection between two pipeline clusters or nodes in the graph. So if a connection between the two exists, irrespective of how many physical wires it includes a single channel is created. The formulation is fairly straight forward:

$$a_j = a_i + w(e_{i,j}) + f_{i,j} + s_{i,j} * (h_{slack} + f_{slack}), \forall v_i, v_j \in V$$

Goal Function is to Minimize $\sum s_{i,j}$

In this equation, a_i, a_j represent the arrival time at node $v_i, v_j \in V$. $f_{i,j}$ is the *free slack* of the channels and is equal to the difference between the TCT and the LCT for the particular channel. $s_{i,j}$ represents the number of slack cells that need to be introduced on the channel so that the circuit is slack-matched. h_{slack} is the *forward latency* of a slack cell, which is a constant that is known at design time and f_{slack} is the free slack of the slack cell, which is also constant and corresponds to the difference between the TCT and the LCT if the slack cell is inserted and depending on the complexity of the circuit can either be precisely calculated or estimated. This is because depending on the design style the LCT of the slack cell could depend on the fanin and/or fanout and/or width of the bus that is routed through it, none of which parameters are known in advance and they actually depend on the solution of the arrival time problem.

This model is fairly accurate for simple pipelines and it has been proven to work well in most cases. However, this formulation does not account for hardware optimizations that could be made and could make this model more accurate. In most design styles, it is possible that several channels leave a particular cluster for several different destinations. However all these channels could be the same wire actually forking to many targets in the netlist. So in this case it is possible to actually share any slack buffers that need to be placed for channels from the same source. The original formulation does not explore this possibility and could think that several buffers are needed in a location that one buffer could be used for all the channels that run in parallel.

In most cases – assuming that we merge the buffers after slack matching anyway – this inaccuracy actually just results in the LP program overestimating the number of buffers needed. However, this problem could be further exaggerated in some cases if the LP formulation concludes that buffers are better placed in a different location. For example if a 2-input cluster has 3 fanout channels due to a wire feeding into 3 other clusters the original formulation would make the LP solver conclude that it is better to place buffers before the cluster than after it, and ultimately use 2 slack buffers instead of 1. This is shown in Figure 14.

A modification was made to the goal function of the LP problem to reflect this inaccuracy from the goal function. The formulation of the arrival times remains identical, but an additional set of parameters is required to be stored and used for evaluation along with some extra constraints that are however linear in terms of the problem size (number of nodes). So again:

$$a_j = a_i + w(e_{i,j}) + f_{i,j} + s_{i,j} * \left(h_{slack} + f_{slack}\right), \forall v_i, v_j \in V$$

But also we add the extra parameters z_i and respective constraints such as

$$s_{i,j} \leq z_i$$

Goal Function is to Minimize $\sum z_i$

And since the minimization of the sums of z_i is the goal function it will be true that $s_{i,j} \le z_i \Leftrightarrow z_i = \max_j \{s_{i,j}\}$. With this formulation, the linear program solver will attempt to find the best solution assuming that it can use a line of buffers that can be shared among all the outgoing channels. This is accurate for cases that slack buffers can support arbitrary fanout and the only cases where it is problematic is situations where a particular design styles poses a hard limit on the fanout of every gate/node. In those cases it might be necessary to have buffer trees rather than buffer lines for slack and in that case our formulation is inaccurate and also optimistic.

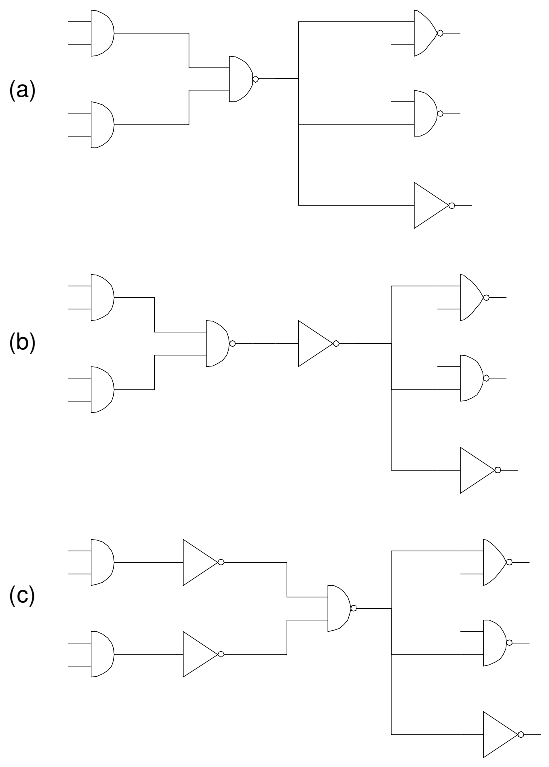


Figure 14 A case where the original slack-matching formulation yields sub-optimal placement of slack buffers. The original circuit in (a) requires on buffer stage. However the old formulation would decide to place the buffers on the input of the gate assuming no sharing (b). The new formulation models the sharing and places the buffer after the gate (c).

4.3 Experiment Setup

In order to test the improvements that can be accomplished by the sharing of the buffers an experiment was setup that focuses on the case of PCHB pipelines. The experiment involved slack matching a netlist that was using the PCHB design style. The formulation was based on the LP approximation described in [5] with the use of a ceiling function. That is the amount of slack per channel is formulated and solved as a real number and the result of the LP problem is rounded up to give the actual number of buffer stages that will be instantiated. This was done because the MILP formulation that is actually accurate, where the slack variables are actually described and solved as integer parameters, is too slow and not practical as also observed in [5] and only very few extremely small examples could be tested.

The objective of our tests was to show the benefits of the buffer-sharing concept while slack matching and characterize the improvements that can be realized. The results are subject to the LP approximation inaccuracy, but have practical value since this approximation allows us to test on larger circuits with tens of thousands of gates, which are closer to real life examples that this method could be applicable to. There are a couple of additional constraints to the PCHB template that was used that slightly affect the accuracy of the results. Those include maximum fanout constraints on gates and an additional requirement that Primary Inputs (PIs) should only fanout to a single gate. When either constraint is violated additional buffers need to be added to yield a functional circuit. Our software was designed to yield functional circuits that can be simulated in Verilog for correctness, therefore those constraints needed to be enforced in order to be able to verify the functionality and performance of the final netlists.

4.4 Experimental Results

For the sake of our experiments we have used 26 examples from the standard ISCAS benchmark circuits as well are 3 of our own examples that represent different sized mathematical operations (2 Multiply Accumulate Units) and a SISO module (Soft-In-Soft-Out used in Error Correcting Decoders in Telecommunication Systems). The results are summarized in Table 4.

		Orig. LP - No Sharing		Orig. LP - Sharing		New LP - Sharing	
Example	**Insts**	**Buffer Area**	**Total Area**	**Buffer Area**	**Total Area**	**Buffer Area**	**Total Area**
c3540	731	6804.17	27225	5235.15	25474.9	3982.69	24121.5
MAC16	1604	11344	48254.1	9893.84	47153.7	8519.73	45508.6
MAC32	1092	8584.7	32071.7	6780.67	30150.1	5696.87	29015.2
s1196	387	4357.32	14964.5	3472.59	13929.1	3388.26	13763.2
s1238	409	4866.05	16109.1	3852.75	14918.9	3693.77	14450.2
s13207	1738	14034.1	52840.9	12908.9	51906.4	8720.18	47919.5
s1423	443	2134.43	12430.5	1835.83	12138.9	1581.47	11901.1
s1488	455	2842.21	14061.8	2021.07	13535.1	1679.62	13275.2
s15850	2229	12408.4	64173.8	11310.8	63170.2	8967.63	60804.9
s27	9	99.5328	299.981	99.5328	299.981	77.4144	277.862
s298	76	597.197	2144.1	503.194	2050.1	470.016	2012.77
s344	82	796.262	2641.77	707.789	2589.24	508.723	2322.43
s349	80	608.256	2430.26	594.432	2437.17	519.782	2368.05
s382	101	829.44	3035.75	707.789	2903.04	663.552	2827.01
s386	89	619.315	2683.24	505.958	2528.41	450.662	2470.35
s400	102	785.203	3002.57	693.965	2907.19	637.286	2822.86
s420	99	1396.22	3533.41	1238.63	3341.26	663.552	2786.92
s444	99	763.085	3015.01	713.318	2956.95	677.376	2894.75
s510	172	1205.45	5374.77	879.206	5213.03	666.317	5011.2
s526	125	873.677	3581.8	821.146	3532.03	837.734	3543.09
s5378	960	7633.61	28001.9	6571.93	27002.4	4813.52	25161.1
s641	135	2092.95	5059.58	1889.74	4788.63	731.29	3642.62
s713	134	2012.77	4954.52	1891.12	4776.19	641.434	3508.53
s820	219	1504.05	6947.94	1241.4	6852.56	740.966	6245.68
s832	213	1492.99	6672.84	1222.04	6498.66	803.174	6097.77
s838	208	2233.96	6903.71	2054.25	6711.55	1194.39	5847.55
s9234	632	3859.66	18323.7	3157.4	17592.4	2742.68	17403
s953	281	1957.48	9277.29	1515.11	8760.27	1338.16	8555.67
SISO	1300	1470.87	28431.8	1426.64	28391.7	1166.75	28126.3

Table 4 Slack matching results for the sharing and no-sharing versions of the algorithm

	Area Savings			
Example	**Buffer vs. Orig. LP - No Sh.**	**Buffer vs. Orig. LP - Sharing**	**Total vs. Orig. LP - No Sh.**	**Total vs. Orig. LP - Sharing**
c3540	41.47%	23.92%	11.40%	5.31%
MAC16	24.90%	13.89%	5.69%	3.49%
MAC32	33.64%	15.98%	9.53%	3.76%
s1196	22.24%	2.43%	8.03%	1.19%
s1238	24.09%	4.13%	10.30%	3.14%
s13207	37.86%	32.45%	9.31%	7.68%
s1423	25.91%	13.86%	4.26%	1.96%
s1488	40.90%	16.89%	5.59%	1.92%
s15850	27.73%	20.72%	5.25%	3.74%
s27	22.22%	22.22%	7.37%	7.37%
s298	21.30%	6.59%	6.13%	1.82%
s344	36.11%	28.13%	12.09%	10.30%
s349	14.55%	12.56%	2.56%	2.84%
s382	20.00%	6.25%	6.88%	2.62%
s386	27.23%	10.93%	7.93%	2.30%
s400	18.84%	8.17%	5.99%	2.90%
s420	52.48%	46.43%	21.13%	16.59%
s444	11.23%	5.04%	3.99%	2.10%
s510	44.72%	24.21%	6.76%	3.87%
s526	4.11%	-2.02%	1.08%	-0.31%
s5378	36.94%	26.76%	10.15%	6.82%
s641	65.06%	61.30%	28.01%	23.93%
s713	68.13%	66.08%	29.19%	26.54%
s820	50.74%	40.31%	10.11%	8.86%
s832	46.20%	34.28%	8.62%	6.17%
s838	46.53%	41.86%	15.30%	12.87%
s9234	28.94%	13.13%	5.02%	1.08%
s953	31.64%	11.68%	7.78%	2.34%
SISO	20.68%	18.22%	1.07%	0.93%
Average	**32.63%**	**21.60%**	**9.19%**	**6.00%**

Table 5 The area savings in terms of slack-matching buffer area and total circuit area realized with the new LP formulation and sharing vs. the other versions.

The results show that the proposed sharing algorithm and the new formulation achieve significant savings over the previous formulation with no sharing as well as over the previous formulation with sharing allowed. There is only one example that the new algorithm did worse compared to the original formulation with sharing. In this particular case the new formulation did worse by just one buffer. This can be attributed to the fact that in both cases the formulations are approximate, in the sense that the results are rounded up to instantiate an integer number of buffers, and in this context an difference of one buffer can be considered a reasonable approximation error.

Overall the sharing of existing buffers seems to achieve average buffer area savings of approximately 32.63% vs. the original LP formulation and no sharing and 21.6% vs. the original LP formulation with sharing. In terms of the total circuit area the savings are approximately 9.19% and 6% respectively on average.

4.5 Other considerations

The new formulation as well as the sharing of buffers have area benefits that are clear from all our experiments. However it is interesting to note a couple of other issues that require careful consideration when choosing which variant to use. Firstly, the new formulation requires additional variables as many as the nodes in the circuit and additional constraints as many as the channels in the circuit, thus making the linear formulation more complex. This results in more memory usage and larger runtime. As the circuits grow in size the new formulation becomes more and more expensive and ultimately impractical. The older formulation can generally tackle larger problems in the same amount of time and thus could be preferential for larger netlists.

Another interesting observation is that the sharing assumes that arrival times of shared buffers can satisfy all leaf nodes attached to them. However this is not always true and as a result sometimes it results in additional buffers being added on a subsequent pass. Without sharing, the arrival times have no such assumptions and thus convergence is achieved usually faster. Both formulations are approximate due to the rounding, so addition of buffers in subsequent paths is always possible, but without sharing it is more rare. This is important when runtime is critical and an additional path might require a very large amount of time that is prohibitive for a given design.

Chapter 5 An Asynchronous ASIC Flow

It is easier to understand the context of this work and the general motivation behind it as part of the entire flow. In this chapter we are going to describe a new flow that has been developed to enable an ASIC design flow for the asynchronous design styles. It will show the commercial tools used and all the customized tools generated and how this flow works from end to end. This is will give the reader a perspective as to what is the value of our work and what its position is relative to the rest of the flow.

Our goal is to generate the framework to enable an asynchronous ASIC flow that will enable engineers to design circuits using asynchronous design styles with the same ease and efficiency as they do for synchronous designs currently. To achieve this we have generated a tool flow that resembles the standard synchronous ASIC flow and in fact reuses most of the existing parts and standard commercial tools. The designer starts with standard HDL input and uses commercial synthesis engines to generate a synthesized image of the logic. We have then designed a tool that takes the synthesized netlist and applies all the design transformation and optimizations that are unique to asynchronous design. After our customized tool is done it generates a netlist back into a standard format so that it can be imported into a standard back-end flow. That enables us to use standard Place and Route, verification and simulation tools that are well known in the industry.

We have also focused throughout this work on the goal of guaranteeing performance. There are three limiting factors for the performance of an asynchronous circuit. The first one is algorithmic loops that were addressed through the distance-based clustering criteria presented in Section 3.2.2. The second one is unbalanced paths and misaligned data in the pipeline, which is a problem solved through slack matching that was discussed extensively in Chapter 4. The third limiting factor is the LCT of a channel, which could make a local handshake the critical path of a circuit. This is addressed in this chapter in Section 5.2.2, where we describe the Clustering algorithm that is implemented as part of this flow.

5.1 Synthesis

The first process in the flow is synthesis. This is done in commercial synthesis tools to convert the design to a synchronous Verilog netlist. The initial synchronous specification is done from a behavioral HDL or RTL HDL, which could be defined in any language that is supported by the commercial synthesis tools. A second input to the tool is the library information. The library information is really an “image” library that abstracts the asynchronous gate details for the synthesis tool (such as whether they are dual-rail or single-rail, their handshaking ports and protocols, etc.), but includes all the information that is useful for it to do a proper design (such as timing, power, area, etc.).

The benefit of this flow instead of trying to generate a new tool is that we can leverage off tools that have been refined through years of research to perform good optimizations, and include predefined and optimized design components such as adders and multipliers that can be used in the data path. So by placing appropriate constraints one could use ripple-carry or carry look-ahead adders to trade between latency and area. Other tasks, such as buffer insertion and gate resizing could be performed as well, by changing the input constraints for the design. In most cases relaxed constraints are enough, since they yield the simplest data path, but if one wanted to shorten the data path due to a large algorithmic cycle time in the final design, modified constraints could be used to force the compiler to generate a different netlist.

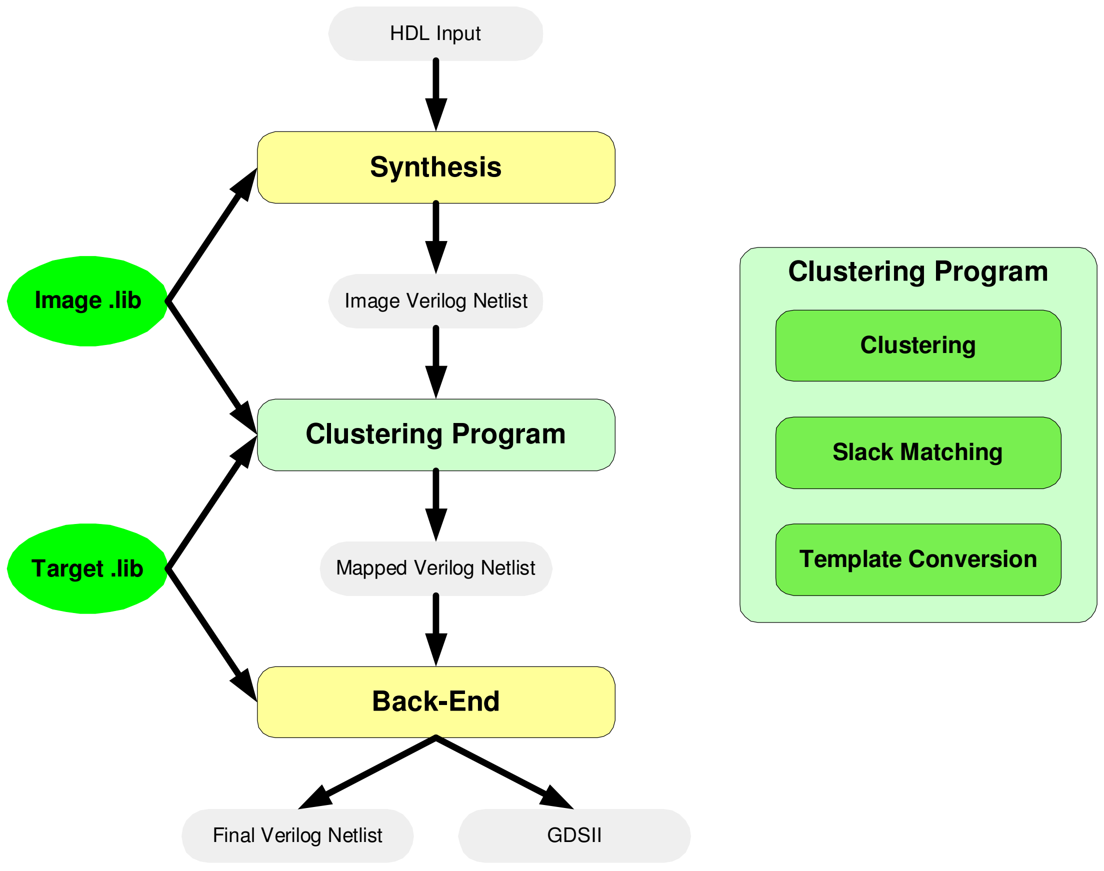


Figure 15 Overview of our ASIC flow.

5.2 The Clustering Program

Most asynchronous design styles allow for multiple gates to be placed within the same pipeline stage. Generally that is done to reduce the area overhead associated with the control logic that is used for handshaking between every pipeline stage and its neighbors. The control logic is large and the more efficiently it gets shared among many gates the smaller the penalty that one has to pay for the asynchronous design style conversion. Depending on the design style the number of gates per pipeline cluster varies widely, but it is believed that the same principles apply in all cases. Also most asynchronous design styles share other design rules, such as limits on fanin/fanouts per pipeline cluster and others. Finally all asynchronous circuits have to obey similar rules about Local Cycle Time, Target Cycle Time and Algorithmic Cycle Time as discussed in Section 2.7.1. And all of these parameters depend on the size and interconnect of the different clusters in the final netlist. So the goal is to minimize the area of the final netlist by grouping gates together as well as possible, but at the same time obeying all the design constraints that are specific to a particular design style and also maintain the functionality and the performance requirements that the designer specified when defining the HDL description of the circuit.

5.2.1 Circuit Representation

Immediately after the original Verilog input file for the circuit is read, the netlist is converted to a generic directed graph structure, with the instances and primary IO converted into nodes and the wire interconnect represented by edges. The original Verilog input file is converted into a generic graph structure that can be manipulate more efficiently during the different optimization operations that are run during processing. Initially the tool reads the entire netlist (flat netlist with no hierarchy) and creates a single pipeline stage out of every gate in the original netlist. The goal of this is to give the tool the most flexibility in terms of clustering, and also start with very simple pipeline clusters that should generally meet any give performance targets more easily.

5.2.2 Clustering

Having formed the fundamental one-gate clusters the next step is to try and group them in larger groups without violating the performance requirements of the design and at the same time preserve the original functionality intended by the user. Our present algorithm does not try to find an optimal clustering solution. Instead it uses a heuristic optimization based on area and performance criteria that implements a steepest-descent-type of algorithm. The main focus of our work so far has been on correctness of the emitted circuit as well as preservation of the performance requirements. The runtime of the heuristic approach is also a big advantage since it allows us to test larger circuits.

The clustering is done by merging two clusters into one (we refer to this as a *local move*) and executing one such move at a time. During each iteration the software looks at all possible moves that are available and executes the one that has the largest performance benefit. It also estimates the area gains from the potential merging, and uses the area benefits to break ties in the case that many moves have the same performance improvements. The area metric is an area estimate of the control logic that can be removed by executing the particular move. The performance metric is an error metric associated with the Target Cycle Time of the circuit and the Local Cycle Time of the individual channels. For each channel an error metric is calculated that is equal to the amount that the LCT violates the GCT. For each move the performance metric is the difference between the sum of errors for all channels before the move and after the move. The largest the metric the more LCT improvements the move will achieve.

After each move the algorithm discards all moves that are no longer feasible and generates new ones from the region of the circuit that was affected and then repeats the process. This avoids costly recalculation of all move data at each step. The algorithm does not select any move that would make the metric worse and thus in practice avoids making LCT worse than the GCT and affecting the performance of the circuit, thus addressing the last and final parameter that could affect performance.

The moves are also checked for other local rules that are associated with particular design restrictions for the design style that is being targeted and are not related to either performance or correctness. For example TOKEN_BUFFER cells are cells that for all templates are initialized during reset with particular data values, unlike regular cells that do not hold data after reset. Those can only be placed in pipeline stages that host similar types of cells. When the clustering algorithm cannot find any candidate local moves to execute it stops. The algorithm is summarized in Figure 16.

```
clustering () {

    generate all allowable moves;
    calculate performance and area metrics for all moves;

    while moves exist, such that performance metric >= 0
    {
        best_move = select_best_move();
        execute(best_move);
        delete all moves no longer allowable;
        generate new moves for affected circuit area;
        calculate performance and area metrics for all new and affected moves;
    }
}

select_best_move()
{
    maxPerfMetric = -inf;
    maxAreaMetric = -inf;
    for every move m
    {
        if (movePerfMetric > maxPerfMetric)
        {
            maxPerfMetric = movePerfMetric;
            maxAreaMetric = moveAreaMetric;
            best_move = m;
        }
        else if (movePerfMetric == maxPerfMetric)
        {
            if (moveAreaMetric > maxAreaMetric)
            {
                maxAreaMetric = moveAreaMetric;
                best_move = m;
            }
        }
    }
}
```

Figure 16 The clustering algorithm that ensures that LCT does not grow.

5.2.3 Fanout Fixes

The circuit is generated using synchronous tools and design libraries. There are certain restrictions in some of the different design styles in terms of the number of fanout gates that each gate will drive. Sometimes this is due to the particular protocol – for example single-track 1-of-N channels (such as those in SSTFB) only support point-to-point connections – or due to performance restrictions (fanout load of the gate or depth of C-element trees required for merging ACK signals). When the given netlist does not obey these restrictions the program has to intervene and correct these faults by instantiating fanout trees for problematic signals using buffer cells, in a process that bears a large amount of resemblance to the buffering and fanout optimization process of the synchronous netlists, only with different criteria and decision metrics. This step is executed if necessary before moving on to create a final netlist.

5.2.4 Final Optimizations

After clustering is done the software also performs slack matching on the netlist. The concept of slack-matching is analyzed in detail in [5], and the tool uses a similar formulation as described there to achieve the performance requested by the user. Some modifications have been made to improve the results based on some practical observations, which will be analyzed later in Chapter 4.

Other local optimizations are done in the netlist during this final stage to ensure the best quality of results. Gate replication is used to reduce the levels of logic in certain cases and buffers are inserted in some cases if it is deemed that the operation might help performance. In general these are minor implementation details that help a practical design, but are probably outside the scope of this work and will not be analyzed as part of this document.

5.2.5 Library Conversion

The next step is to generate a new netlist that instantiates the actual gates that will be used for implementation from the selected asynchronous implementation design. The program also needs to generate and instantiate all the additional control logic that is necessary for the handshaking between the pipeline stages that were defined during clustering and slack matching.

For the design styles that have been imported in the tool so far the final gates that will be used are dual rail gates, even though for other design styles that is not the case. All the nets in the original netlist are in this case converted into pairs of nets each representing the true and false rails of the original single rail signal. The gates are also converted to their dual rail counterparts and the wires are interconnected appropriately. With a dual rail library inversions of inputs and outputs could be handled by swapping the two wires attached to a gate. During this step all gates are converted to their non-inverting counterparts, and the inversion of inputs and outputs takes place by inverting the dual-rail wires. Even though the single rail library uses image gates that include all permutations of possible inversions of inputs and outputs, the final library includes only the non-inverted versions, reducing the amount of library cells that are required for a complete library implementation. Special signals such as power, ground and a global asynchronous reset are also generated. When all the gates have been converted the completion detection trees are formed and the controllers for the pipeline stages are instantiated. The merge and fork cells for the primary inputs and outputs are also instantiated, and the handshaking signals for the left and right environment are added to the top level module.

5.2.6 Verification

The software package finally creates additional information that is required for verification of the resulting design. A header file is written that contains some important parameters for the design, such as the number of inputs, outputs and vectors that are going to be used. Another file that contains a random number generator is also copied to the design directory. Then the software will generate a testbench for the synchronous netlist as well as a testbench for the asynchronous one. A script is also written out that includes the commands needed for running the tests.

The script file commands will first call the random generator module and generate an input file for both simulations that uses random data at all inputs. After this the synchronous netlist is run given the input vectors and the output vectors are recorded at every cycle and sent to a file. The script then executed the command to simulate the asynchronous netlist with the same input file. The results of this run are compared against the results of the synchronous netlist and if they match the testbench will indicate that the test completed successfully. Otherwise it will indicate a failure. At the same time the testbench samples the design outputs and average the number of tokens received over time to calculate the global cycle time in the design.

5.3 Simulation and Back-End Design

The default simulation that our program performs is done in NC-Sim to verify that the translation was successful. However since the netlist is in regular Verilog format, further simulation in any commercial simulator is possible. The Verilog Netlist can also be used as is by commercial back-end tools to perform place and route and verify the performance of the circuit. Back-annotation could also be used for post-layout simulations if desired. However, the flow currently does not support ECO flows and post-layout optimizations. Therefore if the netlist is not yielding the desired results and changes are necessary, the design might have to be processed again from the beginning. Alternatively hand-editing might be able to alleviate the problem if it is easy to identify and fix. Post-layout analysis and ECO-type fixes to the netlist is an interesting and potentially necessary future step.

The flow has been proven in practice to work and we have taken several example designs through, including place & route to produce GDSII. The netlists started from RTL-level code (Verilog) and taken through synthesis, our clustering and slack-matching tool and then Place and Route through Cadence's Encounter tool suite. The results are encouraging showing functional netlists that can successfully place and route and produce functional circuits that would be ready for fabrication. Obviously more is necessary for this flow to reach the levels of sophistication that synchronous flows have reached, but it is a proof of concept that an automated ASIC flow for performance-aware asynchronous design is possible.

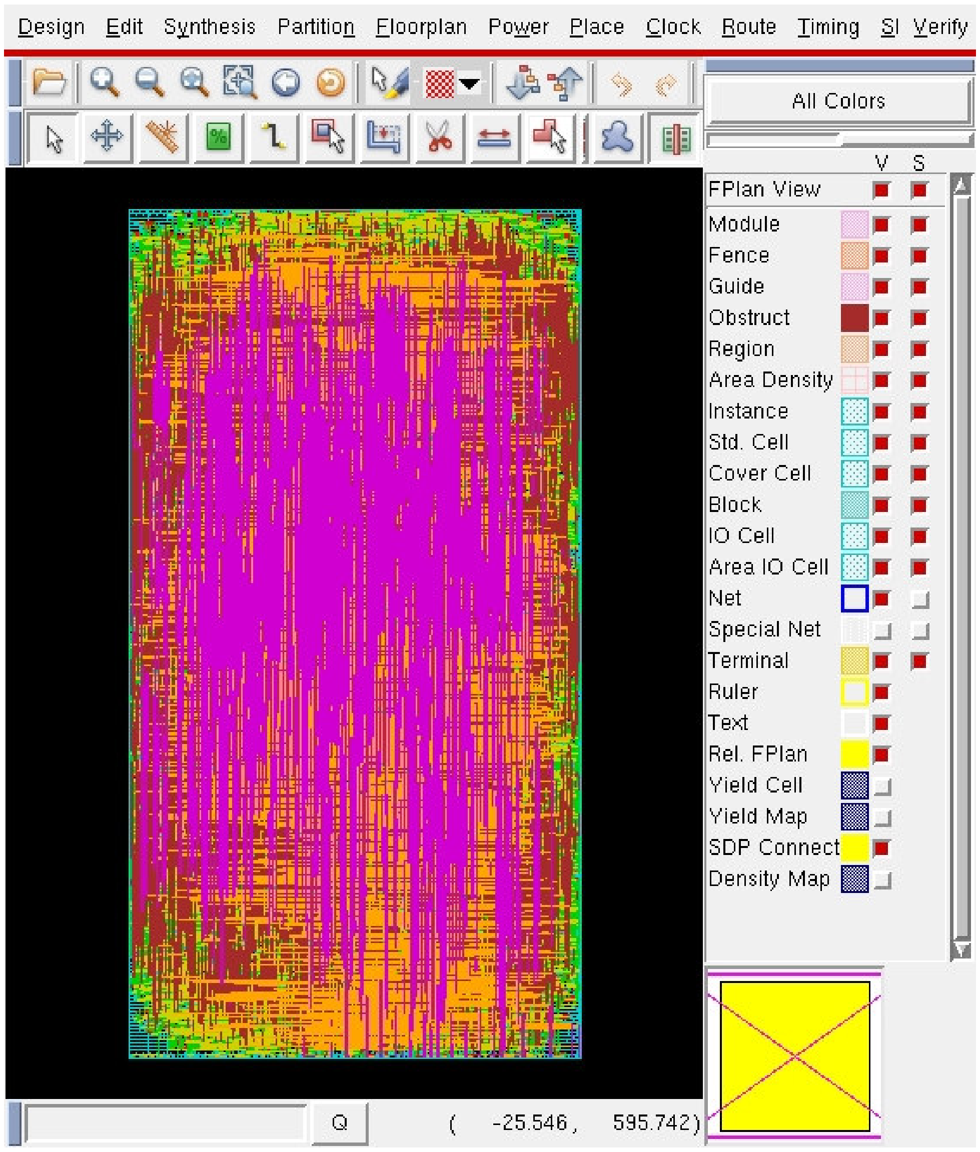


Figure 17 A caption of a finalized placed and routed design after a successful pass through our asynchronous ASIC flow.

Chapter 6 Summary

6.1 Advancements

Asynchronous circuit design has been proposed many times, but has only been adopted in very few isolated cases by the design community. The main reason for this, in our view, is the lack of an automated set of tools that would allow a designer to generate a circuit quickly from a behavioral Hardware Description Language (HDL), just like the ASIC flow that has existed for years for synchronous circuits. As part of this effort we created an automated flow that can automatically generate asynchronous circuits from any HDL using a mixture of custom and existing industry tools.

Due to the fact that asynchronous circuits require a handshaking controller for every pipeline stage, which is used to interface to adjacent pipeline stages, the logic overhead of such circuits is large. By grouping the circuits appropriately one can reduce this overhead and yield circuits that have competitive or even superior characteristics that their synchronous counterparts. Without this grouping asynchronous circuits are most likely going to be far less efficient in terms of throughput per area, even though they might have a substantial absolute performance advantage over their synchronous equivalents.

It is also important to note that asynchronous circuits are data driven in nature. This allows clustering to redefine the pipeline stages irrespective of the pipeline stages defined at the RTL without altering the behavior of the circuit. Thus clustering allows automatic re-pipelining based on performance-driven criteria that can create a circuit that meets the desired performance without changes to the RTL. This makes the design easier and saves designers time and effort that would otherwise be required in adjusting their pipeline stages to meet the required performance.

As we were investigating the tradeoffs early on we discovered that arbitrary clustering with only local constraints had a very high likelihood of yielding non-functional circuits that would deadlock, and very frequently circuits that had performance that was far worse than that of the initial circuit or the desired performance target. Consequently this work was focused on guaranteeing that these two important criteria were maintained by exploring the proper design criteria that would ensure functionality and performance maintenance.

In Chapter 3 we proposed and proved several different criteria that allow us to ensure that the functionality of a circuit is retained through clustering as well as other additional ones that ensure that both algorithmic cycle time and end-to-end latency do not deteriorate. This along with the directed graph model that is proposed is the most important contribution of this thesis and it is an enabling factor for automated pipelining of asynchronous circuits. Using the theory of Chapter 3 we can ensure that the circuits will always work and meet performance goals desired by the designer.

Additionally, our algorithm relies heavily on maintaining a a record of all distances from all the nodes to all the nodes of a graph. We have used the Floyd-Warshall algorithm for deriving the initial distances, but as we modified the graph it was quickly obvious that the algorithm was too complex ($\Theta\left(|V|^3\right)$) to be executed multiple times for updating the distances after every change in clustering. Thus we proposed different practical update algorithms that require far less computation and can be used in practice. One of them was proven to have $O\left(|V|^2\right)$ for circuits with limited fanout. This is an update algorithm that could be useful to a variety of applications that require the maintenance of such an all-pair wise distance array.

Finally Chapter 4 focuses on implementation improvements to the slack matching formulation that was earlier proposed in [5]. The improvements are based on a different formulation for the Linear Program that can be used to solve the Slack Matching problem for a circuit. Even though both methods discussed are approximations of the optimal solution and the model proposed is only a small improvement over the original one the section offers an approach that has practical value. The method discussed along with the new model for sharing slack matching buffers between channels results in overall saving of 13% in terms of total buffering area and approximately 3% in overall circuit area.

6.2 Applications

In this thesis we have created a modeling framework that allows a designer to perform this clustering of the logic gates into pipeline stages by modeling the logic onto an implementation-agnostic graph. This allows us to cluster any arbitrary circuit from its gate-level logic representation into pipeline stages of an asynchronous circuit of any desired design style. Depending on the implementation style the clustering constraints have to be modified to yield a functional circuit that abides by all design rules for the particular style. Area models might also vary quite a bit. But the basic principles of our proposed flow are applicable to all of them.

With the right formulation of the local constraints and appropriate performance and area models for a particular design style one can use this work to design any type of asynchronous pipeline that one desires starting from a gate-level representation of the desired circuit. That means that one could start from any regular HDL representation of a circuit and use a conventional synthesis engine to convert it to gates. Then using this model one could use it to create a pipelined circuit for any design style of their choice. It is believed that this method, since it maintain the functionality and performance of the original circuit, while optimizing its area by clustering, could be used as part of a much broader ASIC flow that would be applicable to all types of asynchronous circuits.

6.3 Open Issues

The focus of this thesis was on maintaining correctness and performance throughout our processing. For all our experiments we used a greedy method for selecting our clustering moves one at a time. This is essentially a steepest-descent method that is most likely far from optimal. Clustering is a hard problem and several heuristic methods have been proposed to address it in different contexts. It is believed that many such methods are also applicable here and it remains an open issue to find practical and good methods for this particular application. Optimization techniques from dynamic programming, simulated annealing, and other optimization techniques are all thought to be applicable for this problem. It is our belief that there deserves to be follow-up work that builds on this framework and will focus now on improving the area results for such circuits, by attempting to find suitable area models and optimization techniques that solve this problem in a good and practical fashion.

6.4 Conclusions

This thesis proposed a method for modeling an arbitrary gate-level circuit using a graph model and methods that allow us to perform area-reducing transformations to it, which are modeled as clustering operations on the graph. The goal is to provide a theoretical foundation for a method that could lead to a fully automated flow for designing asynchronous circuits from an arbitrary gate-level representation that is implementation-agnostic. The long term goal is to provide a framework for further optimizations that could be done to generate asynchronous circuits in an ASIC-like flow that can ultimately compete in performance, area and ease of design with their synchronous counterparts, thus enabling designers to take advantage of asynchronous design techniques in commercial applications. It is our belief that this work is a significant step along a path that can ultimately lead to significant wider adoption of asynchronous design technology.

References

[1]. Enchanced Intel® SpeedStep® Technology. http://download.intel.com/design/processor/datashts/31327801.pdf

[2]. AMD PowerNow!™ Technology. http://www.amd.com/us-en/assets/content_type/DownloadableAssets/Power_Now2.pdf

[3]. A. Bardsley and D. A. Edwards, "The Balsa Asynchronous Circuit Synthesis System," in Proc. Forum on Design Languages, Sept. 2000.

[4]. A. Bardsley and D. A. Edwards, "Synthesizing an asynchronous DMA controller with Balsa," Journal of Systems Architecture, vol. 46, pp. 1309–1319, 2000.

[5]. P.A. Beerel, A. Lines,. M. Davies, N. H. Kim, "Slack matching asynchronous designs", IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'06), 2006,

[6]. K. van Berkel, F. Huberts, A. Peeters. "Stretching Quasi Delay Insensitivity by Means of Extended Isochronic Forks," in Proc. Second working conference on Asynchronous Design Methodologies, pp. 99–106, May, 1995.

[7]. K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-Programming Language Tangram and Its Translation into Handshake Circuits," in Proc. European Conference on Design Automation (EDAC), 1991, pp. 384–389.

[8]. I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "Handshake Protocols for De-Synchronization," in Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Apr. 2004, pp. 149–158.

[9]. T. Chelcea , S. M. Nowick, "Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems," Proceedings of the 39th conference on Design automation, 2002,

[10]. T. Chelcea, A. Bardsley, D. Edwards, and S. M. Nowick, "A Burst-Mode Oriented Back-End for the Balsa Synthesis System," in Proc. Design, Automation and Test in Europe (DATE), Mar. 2002, pp. 330–337.

[11]. C.S. Choy, J. Butas, J. Povazanic, C. F. Chan. "A new control circuit for asynchronous micropipelines", IEEE Transactions on Computers Volume 50, Issue 9, Sept. 2001 pp:992 – 997

[12]. J. Cortadella, A. Kondratyev, L. Lavagno and C. Sotiriou. "A Concurrent Model for De-synchronization." In Handouts of the International Workshop on Logic Synthesis, pp. 294-301, 2003.

[13]. J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "From Synchronous to Asynchronous: an Automatic Approach," in Proc. of International, Design and Test in Europe Conference and Exhibition, 2004.

[14]. J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "De-synchronization: Synthesis of Asynchronous Circuits from Synchronous Specification." in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Oct. 2006.

[15]. K.M. Fant, S.A. Brandt, “NULL Convention Logic^{TM}: a complete and consistent logic for asynchronous digital circuit synthesis” Proceedings of International Conference on Application Specific Systems, Architectures and Processors, Aug. 1996.

[16]. K. Fazel, R.B. Reese; M.A. Thornton, “PLFire: A Visualization Tool for Asynchronous Phased Logic Designs”, Proc. of International, Design and Test in Europe Conference and Exhibition, 2003.

[17]. M. Ferretti, “Single-Track Asynchronous Pipeline Template, PhD Thesis”, University of Southern California, 2004.

[18]. M. Ferretti, P. A. Beerel, “Single-Track Asynchronous Pipeline Templates Using 1-of-N Encoding”, IEEE Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE.02)

[19]. S. B. Furber and J. Liu. “Dynamic logic in four-phase Micropipelines”. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems. IEEE Computer Society Press, March 1996.

[20]. P. Golani, P. A. Beerel, “High-Performance Noise-Robust Asynchronous Circuits,” IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06), 2006, pp. 173-178,.

[21]. J. Kessels and A. Peeters, “The Tangram Framework: Asynchronous Circuits for Low Power,” in Proc. of Asia and South Pacific Design Automation Conference, Feb. 2001, pp. 255–260.

[22]. A. Kondratyev and K. Lwin. “Design of asynchronous circuits by synchronous CAD tools.” In IEEE Design and Test of Computers, vol. 19, no. 4, pp. 107-117, July/August 2002.

[23]. M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, “Asynchronous Design Using Commercial HDL Synthesis Tools,” in Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Apr. 2000, pp. 114–125.

[24]. D. H. Linder and J. C. Harden, “Phased Logic: Supporting the Synchronous Design Paradigm with Delay-Insensitive Circuitry,” IEEE Transactions on Computers, vol. 45, no. 9, pp. 1031–1044, Sept. 1996.

[25]. A. M. Lines, “Asynchronous Pipelined Circuits”, Master’s Thesis, California Institute of Technology 1995, Revised 1998.

[26]. A. J. Martin, “The Limitations to Delay-Insensitivity in Asynchronous Circuits.” Proc of Sixth MIT Conference on Advanced Research in VLSI, ed. W. J. Dally, 263-278, MIT Press, 1990.

[27]. A. J. Martin, “Compiling Communicating Process into Delay-Insensitive VLSI Circuits.” Distributed Computing 1(4):226-234, 1986.

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

[29]. S. Masteller, L. Sorenson, “Cycle decomposition in NCL”, IEEE Design & Test of Computers, Volume 20, Issue 6, Nov.-Dec. 2003

[30]. K. Meekins, D. Ferguson, M. Basta, “Delay insensitive NCL reconfigurable logic”, IEEE Aerospace Conference Proceedings, Volume 4, 2002

[31]. T. Nanya, A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, F. Okamoto, H. Fujimoto, O. Fujita, M. Yamashina, and M. Fukuma. “TITAC-2: A 32-bit scalable-delay-insensitive microprocessor.” In Symposium Record of HOT Chips IX, pp. 19–32, August 1997.

[32]. J. M. Rabaey, “Digital Integrated Circuits A Design Perspective”, Prentice Hall Electronics and VLSI Series, 1996.

[33]. R.B. Reese; M.A. Thornton; Traver, C., “Arithmetic logic circuits using self-timed bit level dataflow and early evaluation”, Proceedings of the International Conference on Computer Design, 2001.

[34]. R.B. Reese; M.A. Thornton; Traver, C., “A fine-grain Phased Logic CPU”, Proceedings of the IEEE Computer Society Annual Symposium on VLSI, Feb. 2003.

[35]. R.B. Reese; M.A. Thornton; Traver, C., “A coarse-grain phased logic CPU”, Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems, May 2003

[36]. R.B. Reese; M.A. Thornton; Traver, C.; Hemmendinger, D., “Early evaluation for performance enhancement in phased logic”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, April 2005

[37]. R.B. Reese; M.A. Thornton; C. Traver, “A coarse-grain phased logic CPU” IEEE Transactions on Computers, July 2005.

[38]. M. Singh and S.M. Nowick. “MOUSETRAP: Ultra-High-Speed Transition-Signaling Asynchronous Pipelines.” Proc. of IEEE Intl. Conf. on Computer Design (ICCD-01), Sept. 2001

[39]. M. Singh, and S.M. Nowick. “High-throughput asynchronous pipelines for fine grain dynamic datapaths”. In Proc. of ASYNC, 2000, pp. 198–209.

[40]. C. Traver.; R.B. Reese.; M.A. Thornton, “Cell designs for self-timed FPGAs”, Proceedings of the 14th Annual IEEE International ASIC/SOC Conference, 2001.

[41]. T.E. Williams. “Latency and Throughput Tradeoffs in Self-Timed Speed-Independent Pipelines and Rings”., PhD Thesis Stanford University, June 1991 Technical Report CSL-TR-90-431

[42]. T.E. Williams, and M.A. Horowitz. “A Zero-overhead self-timed 160ns 54b CMOS divider”. ISSCC Digest of Technical Papers, 1991, pp. 98-296.