Computer Science Department
University of Crete

A Fully-Automated Desynchronization Flow for Synchronous Circuits

Master's Thesis

Andrikos Nikolaos

February 2006
Heraklion, Greece

Πανεπιστήμιο Κρήτης
Σχολή Θετικών και Τεχνολογικών Επιστημών
Τμήμα Επιστήμης Υπολογιστών

A Fully-Automated Desynchronization Flow for Synchronous Circuits

Εργασία που υποβλήθηκε από τον Νικόλαο Ανδρίκο
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
Μεταπτυχιακού Διπλώματος Ειδίκευσης

Συγγραφέας:

_____
Νικόλαος Ανδρίκος, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

_____
Μανόλης Κατεβαίνης, Καθηγητής, Επόπτης

_____
Απόστολος Τραγανίτης, Καθηγητής, Μέλος

_____
Διονύσης Πνευματικάτος, Αναπληρωτής Καθηγητής, Μέλος
Τμήμα Ηλεκτρονικών Μηχανικών & Μηχανικών Υπολογιστών, Πολυτεχνείο Κρήτης

_____
Χρήστος Σωτηρίου, Συνεργαζόμενος Ερευνητής ΙΠ-ΙΤΕ, Επιβλέπων

Δεκτή:

_____
Δημήτρης Πλεξουσάκης, Αναπληρωτής Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Φεβρουάριος 2006

A Fully-Automated Desynchronization Flow for Synchronous Circuits

by

Andrikos Nikolaos

Master's Thesis

Department of Computer Science
University of Crete

Abstract

Variability is one of today's fundamental problems faced by nano-scale electronic circuits, and is posing a very serious threat to the continuation of Moore's law. Alternative design and implementation methodologies, with respect to the conventional synchronous scheme, have been presented in the past for managing timing and environmental variations.

Desynchronization is an approach that converts a post-synthesized synchronous gate-level circuit to a more robust asynchronous one. This work implements the first fully-automated EDA design flow using industrial tools for synthesis, DFT insertion, placement, routing and so on. The flow includes an extra desynchronization step, implemented using a desynchronization tool that automatically applies the desynchronization technique, converting the synchronous circuits and generating scripts for standard tools, in particular to control static timing analysis and physical design.

Two test designs were implemented in a 90nm industrial library down to the mask layout level, in order to validate the technique and compare the desynchronized circuits to their synchronous counterparts. The desynchronized circuits exhibit significantly better variability tolerance, typical instead of worst corner case performance with reasonable area and power consumption overhead.

Keywords: Asynchronous circuits, Desynchronization, Tool automation, Variability.

Thesis Supervisor: Manolis Katevenis, Professor

Thesis Vice-Supervisor: Christos P. Sotiriou, Collaborating Researcher ICS-FORTH

Μία πλήρως αυτοματοποιημένη Ροή Αποσυγχρονισμού για Σύγχρονα Κυκλώματα

Ανδρίκος Νικόλαος

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Η κατασκευαστική μεταβλητότητα είναι ένα από τα κύρια προβλήματα που αντιμετωπίζουν τα σημερινά ηλεκτρονικά κυκλώματα κλίμακας νανομέτρου και αποτελεί μία πολύ σοβαρή απειλή για τη συνέχιση του νόμου του Moore. Εναλλακτικές μεθοδολογίες σχεδίασης και υλοποίησης, σε αντίθεση με το συμβατικό σύγχρονο σχήμα, έχουν παρουσιαστεί στο παρελθόν για την αντιμετώπιση των χρονικών και περιβαλλοντικών μεταβολών.

Ο αποσυγχρονισμός είναι μία μεθοδολογία υλοποίησης, η οποία μετατρέπει ένα σύγχρονο κύκλωμα επιπέδου πυλών σε ένα ασύγχρονο ανάλογο. Αυτή η εργασία υλοποιεί την πρώτη πλήρως αυτοματοποιημένη ροή ηλεκτρονικής σχεδίασης χρησιμοποιώντας συμβατικά βιομηχανικά εργαλεία για σύνθεση, εισαγωγή υποστήριξης κατασκευαστικής δοκιμής, τοποθέτηση, διασύνδεση κοκ. Η ροή περιλαμβάνει ένα επιπλέον βήμα αποσυγχρονισμού, υλοποιημένο χρησιμοποιώντας ένα εργαλείο που εφαρμόζει αυτομάτως την τεχνική του αποσυγχρονισμού, μετατρέποντας σύγχρονα κυκλώματα κατάλληλα και παράγοντας δέσμες εντολών (scripts) για καθιερωμένα εργαλεία, για να υποστηρίξει στατική χρονική ανάλυση και φυσική σχεδίαση με το συμβατικό τρόπο.

Δύο δοκιμαστικά κυκλώματα υλοποιήθηκαν σε μία βιομηχανική βιβλιοθήκη 90nm μέχρι επίπεδο μάσκας, για να επαληθευτεί η τεχνική και για να συγκριθούν τα αποσυγχρονισμένα κυκλώματα με τα αντίστοιχα σύγχρονα. Τα αποσυγχρονισμένα κυκλώματα επιδεικνύουν σημαντικά καλύτερη ανοχή στη μεταβλητότητα, απόδοση στην τυπική αντί για χείριστη περίπτωση με εύλογη επιβάρυνση σε εμβαδόν και κατανάλωση ισχύος.

Επόπτης Μεταπτυχιακής Εργασίας: Μανόλης Κατεβαίνης, Καθηγητής

Επιβλέπων Μεταπτυχιακής Εργασίας: Χρήστος Π. Σωτηρίου, Συνεργαζόμενος Ερευνητής ΙΠ-ΙΤΕ

Acknowledgments

Firstly, I am grateful to my familly for their support all these years. If it was not for their help I would have never made it to here.

I would like to recognize the contribution of my supervisor, Dr. Christos Sotiriou, to the completion of this work and thank him for his guidance, his constructive remarks and the time he devoted.

This work was conducted in collaboration with the ICS-FORTH and funded by it.

Part of this work was conducted in STMicroelectonics, Agrate site, Italy. I am especially thankful to Dr. Davide Pandini and Prof. Luciano Lavagno for their support during my stay there.

Last but not least, I thank all the people with whom I have worked side by side during this thesis. Sharing everyday ideas and experiences has been invaluable.

To the circuits yet to be desynchronized

Contents

1 Introduction **1**

2 The Desynchronization Approach **5**

2.1 General description 5

2.2 Latch Controllers 7

2.3 Flip-flop Substitution 8

2.4 Clock Removal 9

2.4.1 Data Dependencies Graph 9

2.4.2 Controller Network 10

2.4.3 C-Muller Elements 10

2.4.4 Delay of Combinational Logic 11

2.4.5 Network Inter-Connections 12

2.5 Resultant timing 13

3 A CAD Tool for Desynchronization **15**

3.1 Technology Library Support 15

3.1.1 Library Information 16

3.1.2 Extra Latches 16

3.1.3 Latch Controllers 18

3.1.4 Delay Elements 18

3.1.5 C-Muller Elements 18

3.2 Circuit Desynchronization 19

3.2.1 Design Import 19

3.2.2 Automatic Region Creation . . . 19
3.2.3 Flip-Flop Substitution . . . 23
3.2.4 Data Dependency Graph . . . 23
3.2.5 Delay Element Creation . . . 23
3.2.6 Control Network Insertion . . . 23
3.2.7 Design Export . . . 24

4 A Fully-Automated Desynchronization EDA Methodology 25
4.1 Circuit Specification . . . 26
4.2 Synthesis . . . 26
4.3 Design for Testability . . . 27
4.4 Desynchronization . . . 27
4.5 Timing Constraints . . . 27
4.5.1 Latch Enable . . . 27
4.6 Controller Network . . . 28
4.6.1 Loop Breaking . . . 30
4.6.2 Allowing Only Safe Optimizations . . . 30
4.7 Placement and Routing (P&R) . . . 30
4.8 Simulation . . . 32

5 Results 33
5.1 Experimental Procedure . . . 33
5.2 DLX RISC CPU . . . 34
5.2.1 Area Comparison . . . 35
5.2.2 Timing Comparison . . . 36
5.2.3 Power . . . 37
5.3 ARM RISC CPU . . . 38
5.3.1 Area . . . 39

6 Conclusions 41

References 42

List of Figures

2.1 General view of desynchronization flow. 6
2.2 Sample initial circuit. 6
2.3 General picture of a controller. 7
2.4 Desynchronization protocol ordering according to allowed concurrency. 8
2.5 Example of flip-flop loop, where skew cannot be tolerated. 9
2.6 Data dependency graph of the circuit. 9
2.7 Controllers network. 10
2.8 Modeling logic delay with delay elements. 11
2.9 Inner structure of an asymmetric (slow rise, fast fall) delay element. 12
2.10 Modeling logic delay with completion detection techniques. 12
2.11 Fully connected desynchronized circuit. 13

3.1 Flip-flop to latch conversion. 17
3.2 Four-phase semi-decoupled controllers 18
3.3 Grouping algorithm steps. 20
3.4 Grouping algorithm pseudocode. 21
3.5 False logic dependencies induced by buffer insertion. 22
3.6 Grouping based on net buses. 22

4.1 Desynchronization flow. 26
4.2 Clock constraints transformation. 28
4.3 Effective timing view of the desynchronized datapath. 29
4.4 Example of four-phase Desynchronization Controllers 29

4.5 Breaking timing loops within the controllers. 31

5.1 Experimental procedure. 34
5.2 Block diagram of the DLX microprocessor. 35
5.3 Timing results for DLX and DDLX. 37
5.4 Real operation delay comparison between DLX and DDLX. 38
5.5 Power results for DLX and DDLX. 39

List of Tables

2.1 Truth table of a C-Muller element. 10

5.1 Area results for synchronous and desynchronized DLX. 36

5.2 Area results for synchronous and desynchronized ARM. 40

1

Introduction

Manufacturing variability is the uncertainty of predicting the exact properties of a fabricated chip beforehand, *i.e.* at design time [1]. Variability is closely related to varying process parameters, voltage variations, due to IR drops, and temperature fluctuations, thus it has a both a static and a dynamic component. Today, variability is among the most fundamental problems in VLSI circuit design and as the technology constantly scales it is posing a serious threat to the continuation of Moore's Law. This is due to the fact that, although typical corner case may be able to follow Moore's Law, synchronous circuits have to be designed in worst corner case, which deviates more and more from the typical one.

There are two research schools of thought with different proposals for tackling the problem. The first proposes to increase the effort of characterization and analysis of both the design and the libraries. The most representative example is the Statistical Static Timing Analysis (SSTA) methodology [2] where all the variability contributing factors are analyzed statistically and probabilistic results are obtained.

The alternative proposal is to design adaptive circuits which are capable of tolerating parametric

variations. Asynchronous circuits have the natural ability to adapt their timing, thus designing out parametric variability, however their disadvantages, such as being potentially more difficult to design and higher area occupancy, must be tackled. However, one of the key problems related to the adoption of asynchronous techniques is the lack of industrial-quality Electronic Design Automation (EDA) tools capable of supporting their extra complexity of design and implementation and integration with other EDA tools and thus, their adoption by industry is hindered.

Desynchronization [3, 4, 5, 6] is an implementation methodology of producing circuits with a degree of "asynchronicity" using existing EDA tools and flows. Its key characteristic is that it can automatically convert any synchronous circuit to an asynchronous one by replacing the former's clock network with a network of handshaking latch controllers. It has been formally proved [4] that desynchronized circuits exhibit a property known as flow-equivalence [7]. The flow-equivalence property implies that sequences of data of a desynchronized circuit is identical to that of a corresponding synchronous one. As a result, all of the conventional synchronous testing techniques (DFT, simulation, etc.) can be applied in the same way also to desynchronized circuits. Desynchronized circuits are adaptable to variability as their timing is not dictated by an external timing signal, but is manifested through a self-timed latch controller network living inside the chip, which is influenced by the same parametric variations as the circuit itself.

The aim of this master's thesis has been to develop a fully-automated EDA flow for supporting the desynchronized methodology. The new flow is based on an industrial design flows and uses standard tools paired with a desynchronization tool called $drdesync$, which has been implemented from scratch for supporting the desynchronization methodology. $drdesync$ automatically applies the desynchronization methodology through handling i) synchronous circuit conversion ii) the backend timing constraint generation and iii) Static Timing Analysis (STA) automatically. In order to demonstrate the operation of the desynchronization EDA flow, it has been used to implement, down to mask the layout level, two design case studies using the $STMicroelectronics\,CORE9$ 90nm library. Two versions of each design, *i.e.* as synchronous and as desynchronized one, were implemented and compared in terms of area, power, timing and tolerance to variability.

This master's thesis is organized as follows. In Chapter 2 we present and discuss desynchronization. In Chapter 3 we detail the implemented desynchronization tool. In Chapter 4 we describe the desynchronization design flow structure and in Chapter 5 we present and comment on the results obtained for applying the flow to two designs. Finally, Chapter 6 presents the conclusions of this work

along with possible future work.

2

The Desynchronization Approach

In this chapter, the desynchronization methodology is described in detail. A sample synchronous circuit example is used to demonstrate the desynchronization conversion steps.

2.1 General description

Desynchronization is a methodology for converting a synchronous circuit to an asynchronous equivalent. In desynchronization, the global clock network is replaced automatically by a network of intercommunicating latch controllers, whereas the datapath remains intact. This transformation has been shown [4] to preserve a property known as flow-equivalence, which means that each individual sequential element in the desynchronized circuit will possess the exact same data sequence as its synchronous counterpart. This allows for the application of standard synchronous testing techniques.

The main advantage of this methodology is that it can use existing industrial EDA tools, thus the designer is not required to have any knowledge of asynchronous circuits. It is the only methodology for asynchronous circuit design that can use standard libraries and tools starting from HDL specification.

The desynchronization transformation is performed to post-synthesis circuits before proceeding to the backend part of the flow, *i.e.* physical design. An overview of the desynchronization flow can be seen in Figure 2.1 and a full description is detailed in Chapter 4.

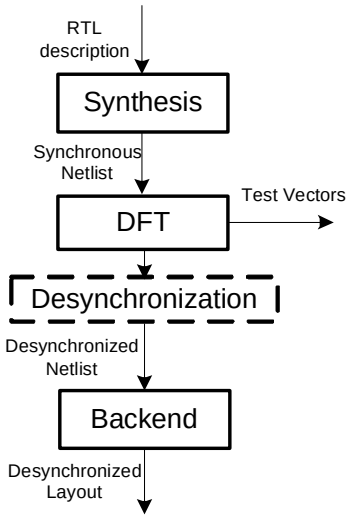


Figure 2.1: General view of desynchronization flow.

Figure 2.2 shows an example of a synchronous circuit. The clouds indicate combinational logic driving flip-flops. The dashed lines indicate the regions of the circuit. By region we define a combinational logic cloud with the flip-flops it drives. The regions can be specified either manually by the designer or derived automatically by the desynchronization tool and their outputs are considered to always be driven by registers.

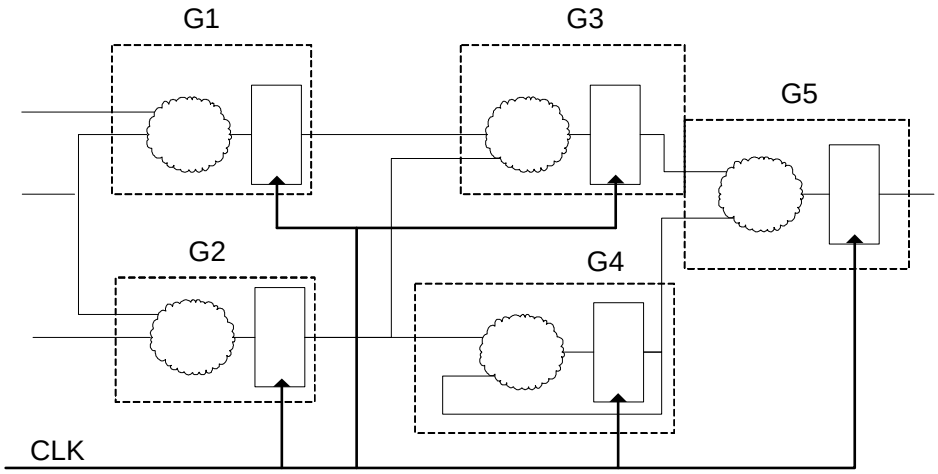


Figure 2.2: Sample initial circuit.

2.2 Latch Controllers

The main element used in the controller network is a latch controller which is an asynchronous circuit implementing a handshake protocol. The general picture of a latch controller can be seen in Figure 2.3. On the left hand side of the figure, the signal `ri`, *i.e.* the input request, indicates that the group of the predecessor controller(s) has (re)finished computing the output data, while the signal `ai`, *i.e.* the input acknowledgement, signals a response to indicate that this group has processed its current data and they can be replaced by new ones. On the right hand side, we have the corresponding signals communicating with the successor controller(s). Thus, signal `ro`, *i.e.* the output request, informs the target controller for the validity of this group's output data, while signal `ao`, *i.e.* the output acknowledgement, indicates that the target group has processed these data. Signal `g`, *i.e.* the latch enable, is used for driving a set of latches, while the signal `rst`, *i.e.* the reset, is used for the controller's initialization.

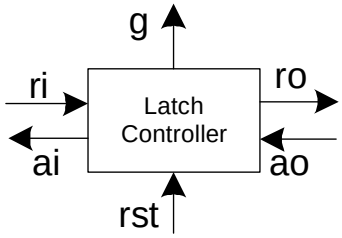


Figure 2.3: General picture of a controller.

For flow-equivalent operation controllers may implement any handshake protocol suitable for desynchronization [4], *e.g.* semi-decoupled, fully-decoupled or desynchronization controller types are all valid. Signal Transition Graphs (STGs) of protocols can be seen in Figure 2.4. STGs are constrained PetriNets [8], which represent the signal dependencies and sequence. In the Figure, signals A and B correspond to two latch controls of two latches in sequence. The protocol implementation can be either 4-phase or 2-phase. This work uses 4-phase semi-decoupled controllers, as they have been shown to exhibit a good tradeoff of signal concurrency and asynchronous circuit complexity [9]. The protocols indicated as $not\ live$ or $not\ flow-equivalent$ in the Figure cannot be used for desynchronization as they will exhibit deadlocks and data overwriting respectively [4].

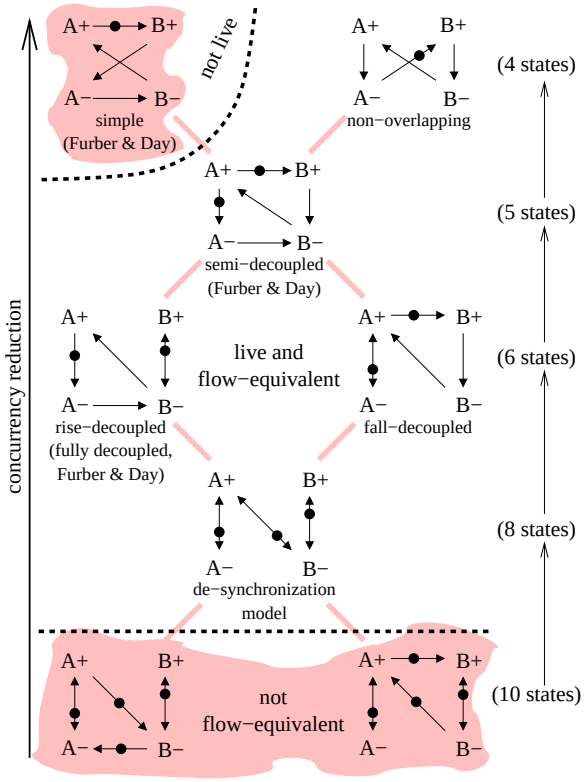


Figure 2.4: Desynchronization protocol ordering according to allowed concurrency.

2.3 Flip-flop Substitution

If the synchronous design is flip-flop based then, for desynchronization to be applied, its flip-flops have to be substituted by latches. A D flip-flop is both conceptually composed of a pair of master-slave latches and this internal structure must be explicitly and practically revealed to maintain equivalent behavior.

This transformation is essential in order to be able to tolerate variable amount of skew imposed at different regions by the controllers. A problematic case can be seen in Figure 2.5. If two flip-flops were to be driven by different latch controllers, skew at their leaves A and B cannot be guaranteed, thus data overwriting will occur.

Also, the conversion of a flip-flop-based circuit into a latch based one can improve performance, albeit area increases. This conversion is not specific to the desynchronization framework only and is known to give better performance even for synchronous systems [10] and, for this reason, it has a value by itself.

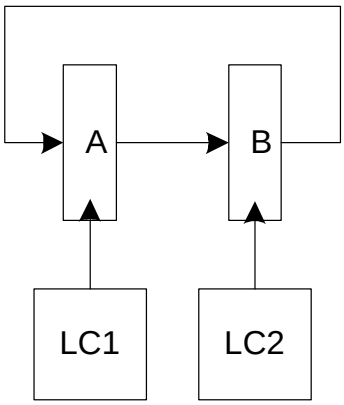


Figure 2.5: Example of flip-flop loop, where skew cannot be tolerated.

2.4 Clock Removal

As mentioned above, the main difference between a synchronous circuit and its desynchronized counterpart is that the former's clock network is replaced by a network of intercommunicating latch controllers, which generate the signals fed to the desynchronized circuit's sequential elements. This section describes what this network is composed of, how it is connected to the synchronous datapath, and how controllers must be connected to each other so that correct circuit operation is ensured.

2.4.1 Data Dependencies Graph

The controller network must respect data flow dependencies among the various parts of the circuit. Thus, the first step in a circuit analysis for desynchronization is to construct a data dependency graph representation of the circuit. In this graph, nodes represent circuit regions and edges data dependencies. Each data dependency between two regions in the circuit, *i.e.* a path from an output of a region to an input of another, is indicated by a directed edge between the two corresponding nodes of the graph. Figure 2.6 shows the dependency graph corresponding to the synchronous circuit of Figure 2.2.

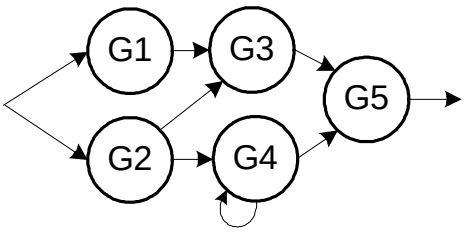


Figure 2.6: Data dependency graph of the circuit.

Inputs	Output
All 0's	0
All 1's	1
Other	Unchanged

Table 2.1: Truth table of a C-Muller element.

2.4.2 Controller Network

The data dependency graph is used for constructing the controller network. Each circuit region, represented by a node in the data dependency graph, will be controlled by a pair of master-slave latch controllers. The master and slave latch pair of a node must be appropriately connected to all its predecessor and successor nodes using synchronization elements (C-Muller gates). Figure 2.7 shows the resulting controller network. For the cases that there are multiple input requests or output acknowledges, C-Muller gates are used as synchronization elements, as shown in the Figure.

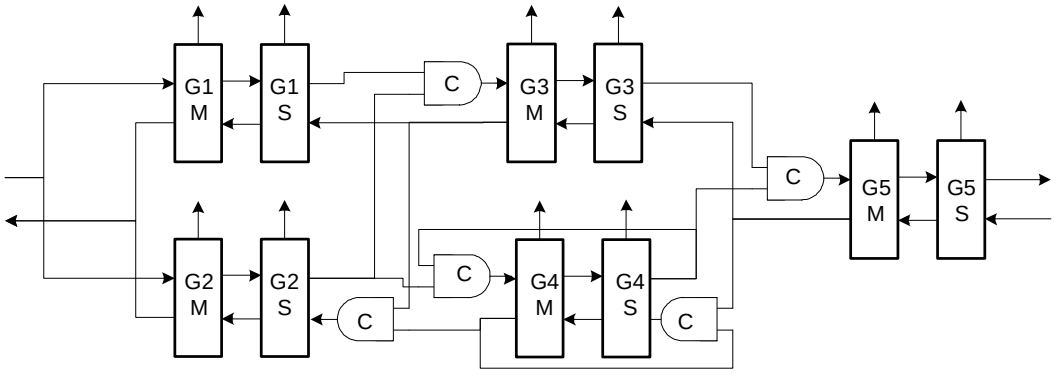


Figure 2.7: Controllers network.

2.4.3 C-Muller Elements

These gates are used to synchronize multiple input requests or output acknowledges. These are the cases when there are many source or target controllers respectively. A C-Muller (or rendezvous) element [11, 12] waits for all of its inputs to be deasserted before deasserting its output and all of its inputs to be asserted before asserting it, thus synchronizing multiple input signals. Its truth table can be seen in Table 2.1.

2.4.4 Delay of Combinational Logic

The desynchronized circuit has to respect setup constraints of its sequential elements. This implies that the combinational logic clouds have to be given enough time to compute their data. Since the request signal is the one that indicate that the logic has finished computing and there are valid data, these signals have to be appropriately delayed for so long as the combinational logic's critical path delay. There are two possible methods to achieve this, *i.e.* using delay elements to mimic the delay of the combinational logic or modifying the combinational logic and embed completion detection.

Delay Elements

The first possible method is the use of delay elements for mimicking the logic's delay. In this approach the request signals pass through a delay element before reaching the target controller. Thus, there is one delay element for each circuit region. Figure 2.8 shows how the delay of delay elements is computed in the general case. The buffers indicate the low-skew buffer trees. The setup constraints are satisfied if $delem\_length + CT\_target \leq CT\_source + CL\_delay$. In the case when the clock trees are balanced or not present. *i.e.* of zero latency, $CT\_target = CT\_source$, the relation becomes $delem\_length \leq CL\_delay$ and the delay element corresponds to the logic's critical path delay.

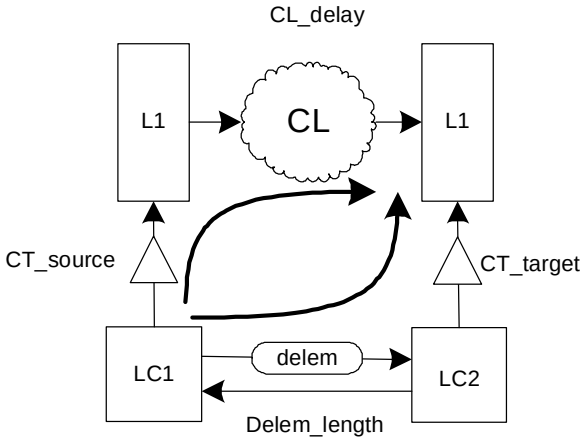


Figure 2.8: Modeling logic delay with delay elements.

The delay elements used are implemented as symmetric in the case of 2-phase handshaking or as asymmetric in the 4-phase handshaking. Symmetric delay elements have the same rise and fall time, whereas asymmetric have high rise time and minimum fall. An example of an asymmetric delay element can be seen in Figure 2.9. A multiplexer may be inserted to configure the final real delay after

layout. In the case of symmetric delay elements the AND gates are substituted by buffers or pairs of inverters.

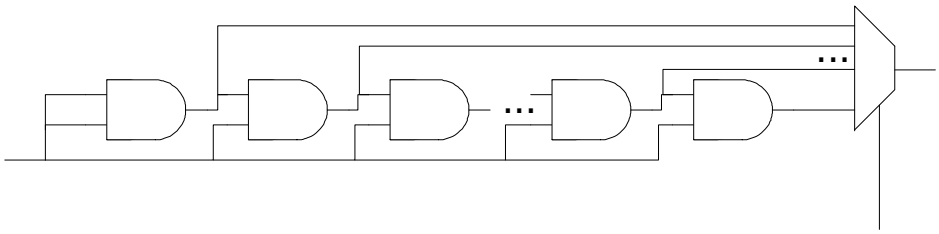


Figure 2.9: Inner structure of an asymmetric (slow rise, fast fall) delay element.

Completion Detection

An alternative to delay elements is to implant completion detection capable logic [13]. In this case, the combinational logic is transformed in a such way that it generates a completion detection signal indicating computation completion and data validity. This signal can then be used as an input request to a controller. This technique has the main advantage that it allows the circuit to operate in actual, average case delay taking into account both the parametric variations and operating data. Its main disadvantage is that the logic transformation induces significant area and power overhead (approximately x2) and thus this approach was not followed in this work.

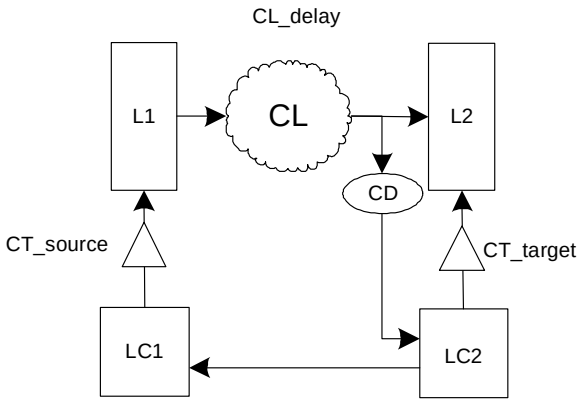


Figure 2.10: Modeling logic delay with completion detection techniques.

2.4.5 Network Inter-Connections

In the last desynchronization step, the entire network of the latch controllers is connected to the original synchronous datapath. The network includes the controller pairs, the C-Muller elements and

the delay elements. The final desynchronized circuit derived from the original synchronous circuit of Figure 2.2 is shown in Figure 2.11. The bold lines indicate the controller network. It can be seen that every region of the original circuit has a combinational logic cloud, a corresponding delay element, one pair of controllers and C-Muller elements used to synchronize multiple input requests or output acknowledgments.

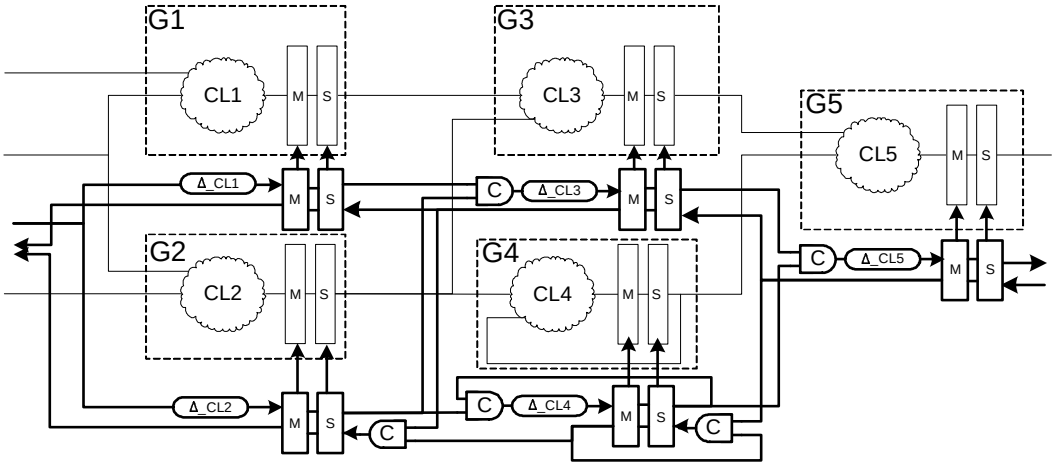


Figure 2.11: Fully connected desynchronized circuit.

2.5 Resultant timing

Desynchronized circuits do not possess external clock signals, as synchronization is achieved internally by the latch controller network. Hence, the timing reference is not external but the result of a self-timed network inside the chip.

Delay elements used are influenced by variability factors, *i.e.* process, voltage and temperature variation, in the same way as the logic they model. This is because they reside in the same chip and special constraints can be used to be placed close to the logic they model. Of course, delay elements must include margins to cope with uncorelated variability, but these margins are far less wider than in the synchronous scheme. Thus, inter-chip process variation and dynamic operating conditions affecting performance can be better tolerated with respect to the synchronous circuits and the circuit's timing can be adjusted automatically and dynamically. Hence, their resultant effective period, is elastic and adaptable to process, voltage and temperature variability factors [14].

In the case of completion detection techniques, the circuit does not only tolerate intra-chip variations but exhibits average case performance as well. Average case relates to the performance de-

pending on the dynamic data of the logic, thus the delay does not always equal to the critical path delay.

3

A CAD Tool for Desynchronization

Since theory is useless without practice, an automated tool was developed to apply the desynchronization methodology. This chapter describes this custom implemented tool called $drdesync$ which enumerates about 10000 lines of C code and numerous scripts for its implementation. $drdesync$ transforms a gate-level synchronous Verilog netlist into a desynchronized one. Before being able to do this special preparation is needed for supporting the target technology library.

3.1 Technology Library Support

Before the desynchronization tool can be used, the technology library must be prepared for it. This has to be done once for each library migration and special effort has been given to be as automated as possible. This phase practically implements all the elements which will be used during circuit desynchronization later. Below there is the description of which is each such element, where it used during the desynchronization conversion and how it is implemented during this phase.

3.1.1 Library Information

The first and most important part of the preparation is the creation the file called $gatefile$ which contains information about the library cells. This file is created using a custom script that parses the $.lib$ standard technology file. For each cell in the library its name, its type (flip-flop, latch, combinational logic gate) its pins, their name and type (input, output, inout, clock) are extracted and put in the $gatefile$. In addition, the $gatefile$ contains replacement rules used during the flip-flop substitution phase of the desynchronization methodology used for replacing the flip-flops with the appropriate pairs of latches. In the cases where the library does not include the required latches new latch names are used and these latches are implemented later as composite modules composed of other standard cells.

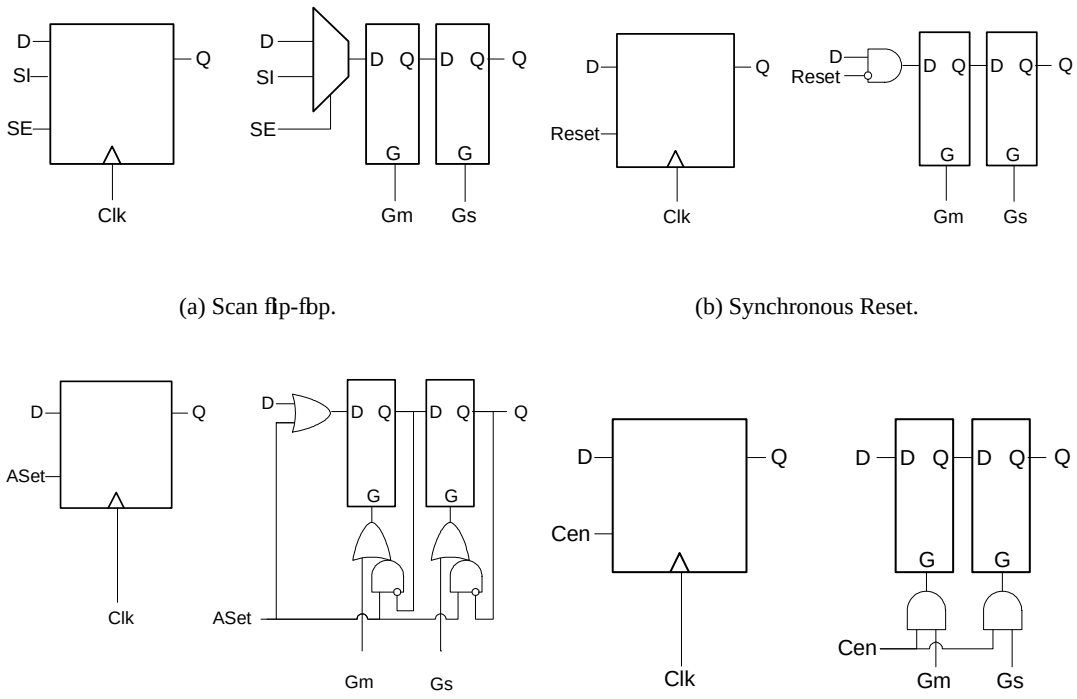
3.1.2 Extra Latches

Generally a D flip-flop is substituted by a pair of simple latches. In the cases of more complex flip-flops the technology library may not include all the required latches which they have to be implemented combining existing latches and extra logic. One such important case may be the absence of scan latches in a technology library for replacing the circuit's scan flip-flops. After the $gatefile$ has been created, a list of all the needed latches is generated and the user implements any missing latches by hand. Figure 3.1 contains some examples of possible cases where latches equivalent of flip-flops must be created. We assume that the only latch included in the library is the simplest possible.

Scan Flip-flops In the cases of scan flip-flops, an extra multiplexer is inserted before the master latch like in Figure 3.1(a)

Synchronous Set/Reset When we have flip-flops with synchronous reset an AND gate with an inverted input is put before the master latch like in Figure 3.1(b). If there is synchronous set instead, we use an OR gate.

Asynchronous Set/Reset The case is more complex when the set or reset signal is asynchronous. In this case we have to take care of two things. First, the latches must open during the asynchronous signal assertion so that the value passes. Second, the latches must close before the signal resets back. Figure 3.1(c) shows the latch circuit that substitutes a flip-flop with asynchronous set. If there is synchronous set instead, we use an AND gate with inverted input before the master latch like in the previous example. Also, the AND cells gating the asynchronous reset



(a) Scan flip-flop.

(b) Synchronous Reset.

(c) Asynchronous Set.

(d) Clock Gating.

Figure 3.1: Flip-flop to latch conversion.

signal do not have inverted input.

Clock Gating The case of clock gating is solved by gating the latch enable signals with an AND gate and can be seen in Figure 3.1(d).

Other Cases Of course this list is not exhaustive. There can also be other examples that they more or less fall in the aforementioned or even hybrid cases that combine two or more of the previous situations.

This transformation models the flip-flop behavior but has a serious disadvantage. Due to increased number of cells the circuit area and power consumption worsen. The ideal case would be to have dual gate latches in the libraries. This would decouple the inner latch enable signals and induce no area or power penalty in the circuit.

3.1.3 Latch Controllers

After all the library information has been gathered and any required extra latch types have been implemented, latch controllers must be implemented for the target library. These are specially designed circuits which need to be hazard-free. Thus, standard logic synthesis cannot be used and the user has to perform technology mapping by hand without decomposing the gates. Figure 3.1.3 shows the 4-phase semi-decoupled controller implementation, used for this work. This circuit has been designed from an Signal Transition Graph (STG) specification in the $petrify$ tool [15]. Any other controller suitable for desynchronization can be used, or even hybrid approaches with various types of controllers.

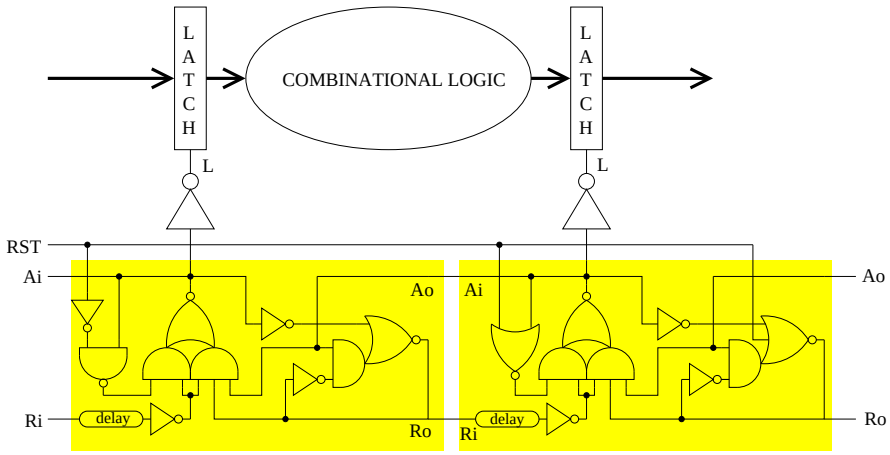


Figure 3.2: Four-phase semi-decoupled controllers

3.1.4 Delay Elements

The combinational logic delay of the combinational logic of each circuit region in this desynchronization flow is modeled using delay elements. During this stage we implement delay elements of variable logic depth, *e.g.* from 1 to 100 logic levels, and perform STA to measure their delay values. Since 4-phase controllers are used, delay elements are asymmetric and each delay element level consists of an AND gate.

3.1.5 C-Muller Elements

Finally, the only elements used during the desynchronization conversion of the circuits yet to be implemented are the C-Muller elements for multiple inputs. Multiple input, *e.g.* from 2 to 10 bits, C-Muller elements are implemented by describing in Verilog HDL and then synthesizing with a conventional

synthesis tool.

In this point, the technology library has been adapted to $drdesync$ and desynchronization of a circuit may be performed.

3.2 Circuit Desynchronization

This section describes how $drdesync$ performs the automatic circuit desynchronization. The tool has a command line interface and the desynchronization operation consists of a sequence of steps. A typical sequence of steps is presented below.

3.2.1 Design Import

The tool supports the full gate-level Verilog specification and reads designs in this format. During design import, escaped names are substituted by simple ones and assign statements are replaced wherever possible. These modifications produce a cleaner netlist without altering the design functionality. After the synchronous netlist has been imported, it is ready to be desynchronized

There are two main advantages with the fact that $drdesync$ operates in gate-level as opposed to Register Transfer Level (RTL). Firstly, not always can be assured that the RTL specification of the design is available, whereas a gate-level specification can be generated even from final layout. Secondly, in this level, the circuit is already logically optimized and thus only the true data dependencies are kept. Hence, the algorithms performing the automatic grouping can produce better results.

3.2.2 Automatic Region Creation

An essential part of the desynchronization flow is the identification of the desynchronization circuit regions. These regions contain combinational logic clouds with their driven flip-flops and for basic version of the desynchronization methodology these logic clouds need to be independent, *i.e.* no connections between logic clouds of different regions are allowed. This specification has to be automatic because there are situations where the designer is not able to specify the regions by himself, *e.g.* the case of a third party design which may be hard to read. Also, even if he is able to do this, he may not partition it according to the actual data dependencies but according to the high-level architectural view for example.

In this step of the flow, it is possible for the designer to choose whether the automatic grouping algorithm will be used or the regions will be specified manually. If the automatic grouping stage is skipped, the design has to comply with a certain form which requires a two-level netlist in which the

top level module contains only flattened submodules considered as the circuit regions.

The automatic identification of the circuit regions is the most difficult part of the desynchronization procedure since we needed to implement an algorithm solving a partitioning problem. The algorithm used finds the independent combinational logic clouds, taking into account the connections between combinational logic gates.

Grouping Algorithm

The algorithm is based upon the connections between the circuit gates. A high-level view can be seen in Figure 3.3 whereas algorithm's pseudocode is presented in more detail in Figure 3.4.

During the first step of the algorithm only connections starting from logic gates are taken into account and each connected component is considered as one region. This includes the logic gates along with their directly driven flip-flops. In the second step, any ungrouped flip-flops driven by already grouped flip-flops are added to the latter's region. In the final step, any remaining sequential elements. *i.e.* flip-flops registering circuit inputs are grouped together in a new region. In addition to the main idea described above, there are some additional features taking into consideration special cases.

1. Group together all the combinational gates connected to each other.
2. Add to each group the sequential elements driven by the group's members.
3. All the rest sequential elements are assigned to the extra Group 0.

Figure 3.3: Grouping algorithm steps.

Logic Cleaning

Since the grouping algorithm is based on the logic connections, we want these connections to correspond to real data dependencies so that better results are produced. Thus, the netlist has to contain only "clean logic", *i.e.* free of buffers or pairs of inverters, added for signal buffering by the synthesis tools. Before running the actual algorithm, these cells are removed so that they do not infer any logic dependencies between the combinational gates. This is shown in Figure 3.5.

In the cases of In-Place Optimization (IPO) flow the removed logic does not need to be put back, since it adds no real functionality and the optimization phases during the backend part of the flow deal with any needed buffering insertion. If the designer wishes to use the automatic grouping but also

```
CONNECTIONGROUPING(Module)
 1  # First step
 2  for each Ungrouped_Combinational gate in ModuleInstances
 3  do
 4     Group = newGroup()
 5     Group.add(gate)
 6     for each unvisited* cell in Group
 7     do
 8        Group.add(cell → CombinationalSourceCells)
 9        if cell is Combinational
10          then Group.add(cell → TargetCells)
11
12        for each bus in cell → TargetBuses
13        do
14           Group.add(bus → SourceCells)
15
16  # Second step
17  for each Grouped_Sequential gate in ModuleInstances
18  do
19     Group = gate → Group
20     for each unvisited* cell in Group
21     do
22        Group.add(cell → SequentialTargetCells)
23
24  # Third step
25  for each Ungrouped gate in ModuleInstances
26  do
27     Group0.add(gate)
```

* unvisited refers to each **for each** loop independently

Figure 3.4: Grouping algorithm pseudocode.

wants the circuit to remain intact during the backend phase, then he has to follow the next steps. First, the circuit is automatically “cleaned” and grouped. Then, it is imported again in the synthesis tool and any needed buffer insertion takes place. Finally, it is imported in the desynchronization tool and

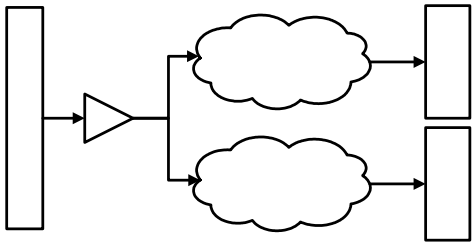


Figure 3.5: False logic dependencies induced by buffer insertion.

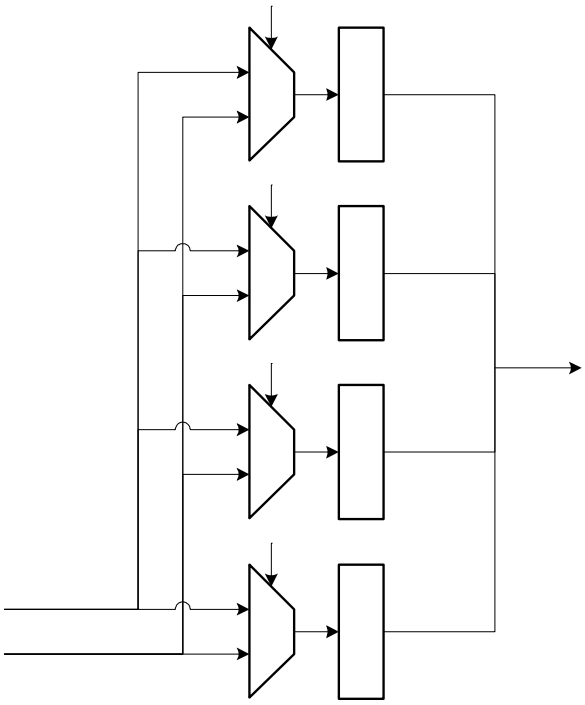


Figure 3.6: Grouping based on net buses.

continues to the rest of the desynchronization process skipping the grouping phase.

Buses

Another heuristic which is used is by-name grouping for buses. Buses can be determined only by name and this rule can be used only if the synthesis tool has not collapsed the bus in individual nets, *i.e.* $bus[n]$ versus bus_n naming. Its importance resides in the fact that buses are usually driven by flip-flops or logic gates that the designer considered to be handled collectively. One such example can be seen in Figure 3.6 where a multibit multiplexer would otherwise be splitted in multiple groups.

Flip-flop to Flip-flop Direct Connections

Flip-flops directly driven by other flip-flops are grouped together since their only purpose is to store history of signals for a number of cycles. This heuristic is implemented during the second step of the algorithm when, ungrouped flip-flops directly driven by already grouped flip-flops are added to the latter's corresponding region.

False Paths

Finally, the designer can mark net connections to be ignored when they refer to false paths. One such case is global signals being connected to the whole circuit, *e.g.* synchronous reset, or clock gating signals. This marking is specified via the tool's command line.

3.2.3 Flip-Flop Substitution

The desynchronization technique requires a latch design to ensure that no data overwriting occurs. Thus, the flip-flops in the circuit are substituted by pairs of latches of equivalent behavior according to the rules in the $gatefile$. The transformation is possible using also the extra latches implemented during the library integration.

3.2.4 Data Dependency Graph

After all the circuit regions have been specified and the flip-flops substituted, the data dependency graph of the circuit is constructed following the rules described in Paragraph 2.4.1. It is reminded that the nodes of this graph correspond to the circuit regions, whereas its directed edges correspond to the data-dependencies between these regions.

3.2.5 Delay Element Creation

For each circuit region we compute the critical path delay of its combinational logic cloud. The delays are computed by exporting the design to a file and then invoking any tool able to perform STA. The values computed are used to choose the delay elements with the appropriate length according to the values computed during the library integration. Also, the designer may specify in the tool's command line any wanted delay element multiplexing, which is implemented automatically.

3.2.6 Control Network Insertion

This is the final and main phase of the desynchronization technique. For each group, the count of incoming and outgoing edges is computed. These numbers equal to the fanin and fanout of the cor-

responding node of the data dependency graph. Then, these numbers are exported to an external tool which decides and implements the appropriate controller, which is then read back in the tool. The controllers along with the delay elements implemented in the previous step are connected to whole circuit. After this, the desynchronized circuit is ready for the backend phase of layout generation.

3.2.7 Design Export

After all the phases are completed, the desynchronized design is ready to continue in the backend flow. The tool exports the desynchronized design along with physical timing constraints. The gate-level netlist is in Verilog format, but BLIF format for exporting to SIS tool [16] is also supported.

4

A Fully-Automated Desynchronization EDA Methodology

This chapter describes the EDA methodology for desynchronization, which uses conventional synchronous Hardware Description Languages (HDL) and tools. It begins with a synchronous circuit specification and generates desynchronized mask layout. Initially, circuit synthesis takes place generating gate-level netlist. Next, Design for Testability (DFT) insertion is used in order to be able to test the design after manufacturing. Later, the desynchronization step takes place, desynchronizing the netlist. Then, during the backend phase, the placement and routing of the circuit is performed and final mask layout is generated. Finally, behavioral simulation is used to verify the correct circuit functionality. A general view of the flow can be see in Figure 4.1.

The desynchronization flow is compatible with any conventional synchronous EDA flow, and the only change is the desynchronization step insertion before the backend phase. No specific assumptions are made about the tools used for the synthesis or backend parts.

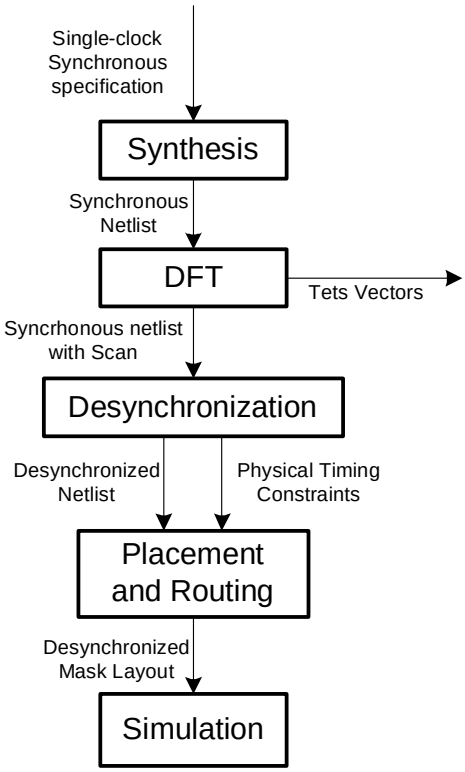


Figure 4.1: Desynchronization flow.

4.1 Circuit Specification

The circuit has to be specified in a HDL. Also, all the power and timing constraints corresponding to the intended circuit application have to be provided. The desynchronization EDA flow's purpose is to generate a desynchronized mask layout. This layout must behave equivalently to the initial circuit specification and satisfy the constraints given. Currently, the desynchronization flow supports only single clock circuits.

4.2 Synthesis

The first step of the desynchronization EDA flow is the synthesis stage which may be performed using any conventional synthesis tool. Initially, the logic synthesis of the synchronous circuit specification is performed. This results to the translation of the RTL specification into boolean logic functions. Then the technology mapping phase follows which maps the boolean logic functions to the desired technology library. Finally, the gate-level netlist is generated.

4.3 Design for Testability

Chips must be capable of being tested also after fabrication. Hence, after synthesis, there is the DFT phase where all the sequential elements, *i.e.* flip-flops or latches, are substituted by scan ones connected in a scan chain, for making the circuit observable. After the scan chain insertion the test vectors are extracted. These vectors are used after fabrication to detect any chip errors. The netlist with the scan chain is then passed to the desynchronization tool.

4.4 Desynchronization

The desynchronization step takes place using the $drdesync$ tool. The input to the tool is the synchronous post-synthesized gate-level Verilog netlist and its output corresponds to a desynchronized netlist which is ready to go through the rest of the backend flow. Also, the tool generates all the extra timing constraints needed because of the desynchronization transformation. An important issue is that $drdesync$ uses standard file formats (Verilog for netlist, Synopsys SDC for timing constraints, etc) and thus, it may be embedded to virtually any modern industrial EDA flow.

4.5 Timing Constraints

The desynchronized circuit has the exactly same datapath as its synchronous counterpart, however i) it is a latch design and ii) it contains an asynchronous controller network. Thus, its physical timing constraints are stricter. Below, it is described how we managed to constrain both the controller network and the original datapath.

4.5.1 Latch Enable

As flip-flops are substituted by latches, original clock signal descriptions must be changed. For the clock input signal in the original design two non-overlapping signals must be specified for the corresponding master and slave latches respectively. Each of these is driven by multiple source pins, all the master and slave enable signals of the controllers network respectively.

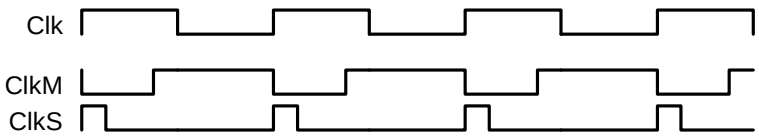
Figure 4.2 shows an example of such timing constraint transformation and the resulting timing relation between the signals. The period is the same as with the original clock. The falling edge of the master signal and the rising of the slave coincide with the rising edge of the original clock. This is the equivalent behavior when considering the master-slave nature of a D flip-flop. The other edges are not that important expect of course the fact that the signals have to be non-overlapping. Although during the circuit operation the generated enable signals may overlap, this does not pose any thread since this

```
create_clock -name "Clk" -period 2.4 -waveform {0 1.2} [get_ports clk]
```

(a) Clock specification

```
create_clock -name "ClkM" -period 2.4 -waveform {1.0 2.4} [get_pins {*_Ctrl/core/master/g_out/Z}]
create_clock -name "ClkS" -period 2.4 -waveform {2.4 2.8} [get_pins {*_Ctrl/core/slave/g_out/Z} ]
```

(b) Master-Slave latch enable specification



(c) Resulting timing relation

Figure 4.2: Clock constraints transformation.

occurs only when it is safe for the circuit, guaranteed by the controller network. Hold constraints are automatically satisfied since we have a latch design and sufficiently wide pulses.

Specifying the enable signals like this, allows for the backend tool to consider the datapath independently of the controller network. Figure 4.3 shows what is the effective view to the backend tool. In this way, it is ensured that the datapath will be optimized using the same approach as with a synchronous version. This method of specifying the enable signals takes into account also any differences in the depth of low-skew buffer trees of each enable signal, as these will be matched by the Clock Tree Synthesis (CTS) algorithm of the backend tool. The low-skew buffers in the previous Figure are indicated by buffers.

Alternatively, the buffers can by bypassed by specifying the latch enable pins as source pins for the enable signals. In this case, any buffer tree depth difference must be modeled in the delay elements as described in Paragraph 2.4.4.

4.6 Controller Network

The controller network is a fully asynchronous circuit, thus it contains cycles, which can be a problem for conventional EDA tools, *e.g.* when running Static Timing Analysis during buffering and physical design. Figure 4.6 shows the semi-decoupled 4-phase latch controller implementation [4, 17] used in this work.

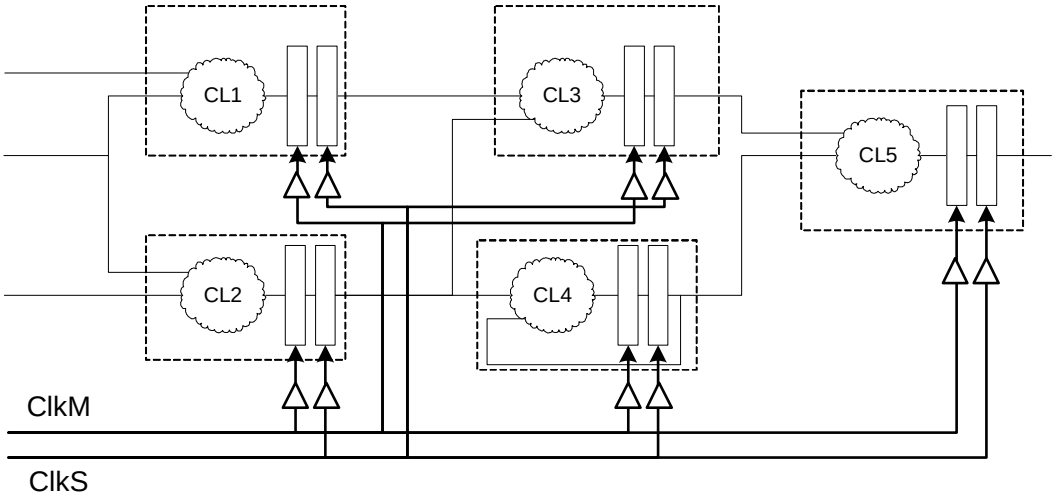


Figure 4.3: Effective timing view of the desynchronized datapath.

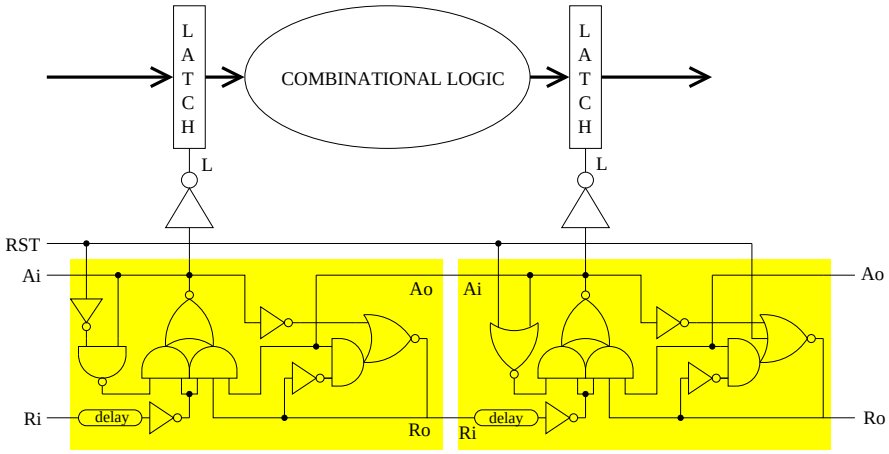


Figure 4.4: Example of four-phase Desynchronization Controllers

When STA is performed, any cycles in the combinational netlist must be broken, *i.e.* some edges must be removed. Such edges can be, for example, those classified as back-edges by the STA graph traversal algorithm. The definition of a back-edge during the algorithm execution depends on the order of graph traversal. Thus, the places where the graph is cut are arbitrary with respect to the designs functionality. In the synchronous case, this is safe, since feedback loops in combinational logic can never be sensitized.

For an asynchronous circuit, such as the one shown in Figure 4.6, cutting the cycles at arbitrary locations may imply a significant performance penalty if the cut edge is part of the critical cycle [18]

of the circuit. The critical cycle of an asynchronous circuit is one whose delay determines the performance, *i.e.* effective period, of the circuit. If a connection in the critical cycle remains unconstrained, during the execution of the timing-driven P&R algorithms it may easily become very slow seriously affecting overall performance.

4.6.1 Loop Breaking

In this work the loops in the controllers had to be cut by hand, making sure that critical loops are fully constrained. This procedure has to be performed only once for each controller implementation.

Firstly, the critical path in the controller network has to be found, which is a loop traversing multiple gates. The portions of the critical path for a full rise/fall period can be seen in Figure 4.5(a). Most gates are included twice in the path. Then, all these connections are combined together and the minimum graph to constraint is obtained (Figure 4.5(b)).

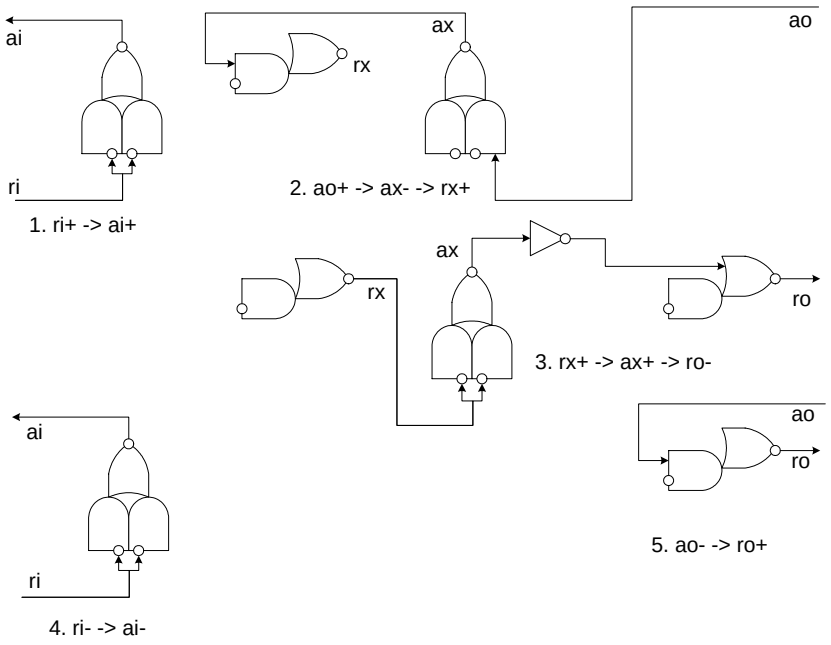
Since this graph needs to be acyclic to perform STA, one more gate pin has to be timing-disabled (the X in Figure 4.5(b)) so that all the timing loops are broken. Fortunately, this specific gate can be constrained through its other pins. Finally, some other connections not creating loops are also allowed and the final acyclic timing graph is produced. Figure 4.5(c) shows the whole internal circuit of the semi-decoupled controller sued in this work with the X's showing the timing-disabled pins. In all the paths remaining min/max input-to-output timing constraints were used, so that the timing-driven algorithms could optimize also the controllers network.

4.6.2 Allowing Only Safe Optimizations

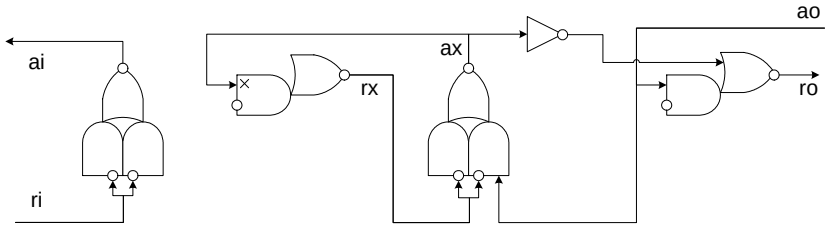
The controllers are specially implemented circuits to guarantee that the desynchronized circuit functions equivalently to its synchronous counterpart, thus their signal transitions have to be hazard-free. Allowing any logic re-synthesis algorithm to decompose the controller gates may potentially create hazards. Hence, the gates within the controllers are specified as "size_only", so that the Placement and Routing (P&R) tool can optimize the circuit by gate resizing and buffer insertion, but not using any re-synthesis techniques.

4.7 Placement and Routing (P&R)

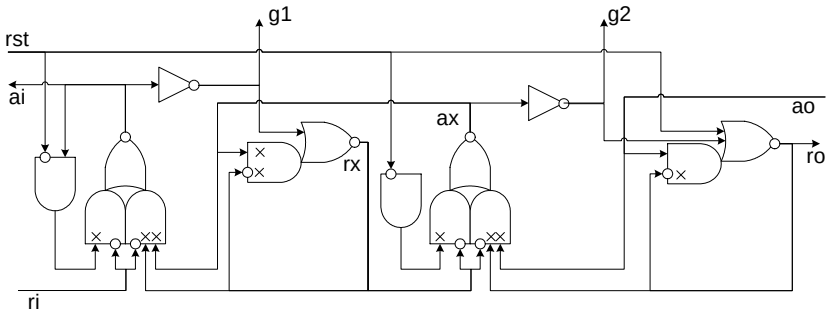
After all the constraints and the overall specification is completed and checked, the design is ready for the backend stage of P&R. In this phase the circuit gates are placed, low-skew buffer trees are inserted and the circuit nets are routed. During all these steps special emphasis is given to satisfy the



(a) The controllers' critical path portions.



(b) The collective graph with the critical portions.



(c) Semi-decoupled controller circuit with timing-disabled pins.

Figure 4.5: Breaking timing loops within the controllers.

constraints. Finally, the final layout is generated.

4.8 Simulation

After all the final layout is ready, functional simulation is performed to verify the correct behavior of the design. The simulation uses delays derived from the actual layout with full parasitic extraction. It is worth noting that testbenches for the desynchronized versions are almost identical to those for the synchronous designs. The only change needed is the replacement of the clock references by corresponding request/acknowledge signals due to the flow-equivalence property.

5 Results

This chapter describes the experimental procedure to evaluate the EDA desynchronization flow. In order to do this two designs were used. They were implemented in a 90nm fabrication process, using a conventional synchronous and the desynchronization design flow respectively. The two versions of each design, *i.e.* synchronous and desynchronized, were then compared in terms of layout area, equivalent frequency and power consumption for both best and worst case conditions [1]. The results acquired are presented and commented.

5.1 Experimental Procedure

This section describes the exact procedure followed to evaluate the desynchronization flow. Two versions of each design were implemented by using the `Synopsys-Avanti STMicrolectronics` in-house synchronous design flow. We used the flow as is for the synchronous version, whereas we integrated it to the desynchronization flow for the desynchronized version. Both synchronous and desynchronized flows use the exact same tools and libraries so that a fair

[1] The library does not include typical case conditions

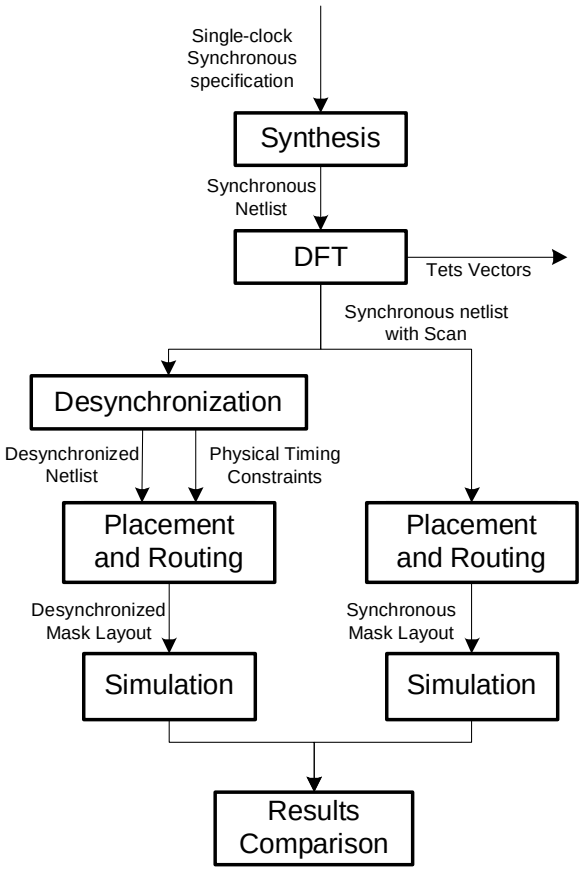
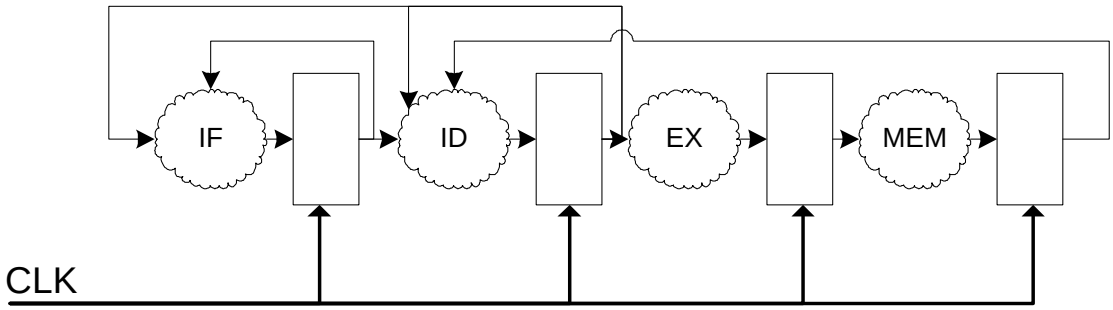


Figure 5.1: Experimental procedure.

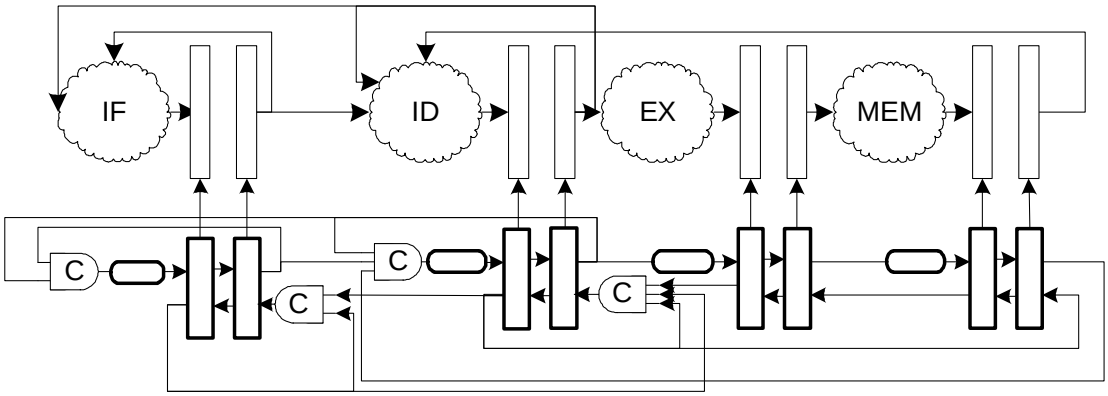
comparison can be made. A high-level view of the whole procedure can be seen in Figure 5.1. The tools used are `Synopsys Design Compiler` for logic synthesis and scan insertion, `Synopsys PrimeTool` for design and constraints checks, `Synopsys Astro` for P&R, `Mentor Graphics Calibre` for Design Rules Check (DRC) and Layout Versus Schematic (LVS) checks and `Cadence VerilogXL-Simvision` environment for functional simulation.

5.2 DLX RISC CPU

The first design used is an implementation of the RISC DLX microprocessor [19] which is widely used in academia. DLX has been implemented in Verilog HDL, supports the full DLX integer ISA and it has no data forwarding between the pipeline stages. The automatically assigned desynchronization regions in this case matched the 4 pipeline stages of the processor. The block diagrams of the DLX design, for both synchronous and desynchronized versions, can be seen in Figure 5.2. The target library has been the High-Speed version of the `ST CORE9` 90nm library.



(a) Synchronous version.



(b) Desynchronized version.

Figure 5.2: Block diagram of the DLX microprocessor.

5.2.1 Area Comparison

The area results were taken from the Astro tool used for Placement and Routing (P&R). Also post-synthesis results are presented so that extra comparisons are possible to be made. Table 5.1 presents the area results for the two versions of the DLX processor with DDLX referring to the desynchronized one. The total area overhead (core size) of the desynchronized circuit compared to the synchronous one is about 13.5%. We can notice that this overhead is mainly coming from the flip-flop substitution by latches (+17.66%) meaning that it can be significantly improved if the target libraries contain two-clock flip-flops or if their latches are more optimized.

Design Phase	Area Property	DLX	DDLX	
				% Overhead
Post Synthesis	# nets	14925	16636	11.46
	# cells	14855	16550	11.41
	cell area (μm^2)	188321.49	200593.14	6.52
	combinational logic (μm^2)	134443.56	137200.78	2.05
	sequential logic (μm^2)	53877.93	63392.36	17.66
Post Layout	# nets	15016	16783	11.77
	# cells	14951	16781	12.24
	standard cell area (μm^2)	196951.34	214266.98	8.79
	core size (μm^2)	207195.54	235048.18	**13.44**
	core utilization (%)	95.06	91.16	4.10

Table 5.1: Area results for synchronous and desynchronized DLX.

5.2.2 Timing Comparison

Comparisons about speed performance were made by taking simulation results for both the two design versions. The desynchronized version included 8 input multiplexed delay elements so that we are able to calibrate their length. The multiplexers selection setup was the same for all the delay elements.

Figure 5.3 presents the different results in relation to each value used for the multiplexer inputs. The dashed lines refer to the setups in which the delay elements become too short for the logic they match. The most important thing to note is that this happens on the same point (delay selection 2) for both best and worst case. This shows that the delay elements follow the logic in the same way for both the extreme corner cases which indicates that they should also follow it for every case in between.

Typical Case Operation

If the best working setup is taken into consideration (delay selection 2) and if we operate both synchronous and desynchronized version in the worst case setup then the overhead is about 20%. This corresponds to the 3 complex gates control overhead between the the falling edge of the slave enable signal to the rising edge of the master one, while the DLX critical path includes 13 levels. For designs with longer critical paths this overhead is lower.

However, one must keep in mind that a fundamental motivation to move to asynchronous design

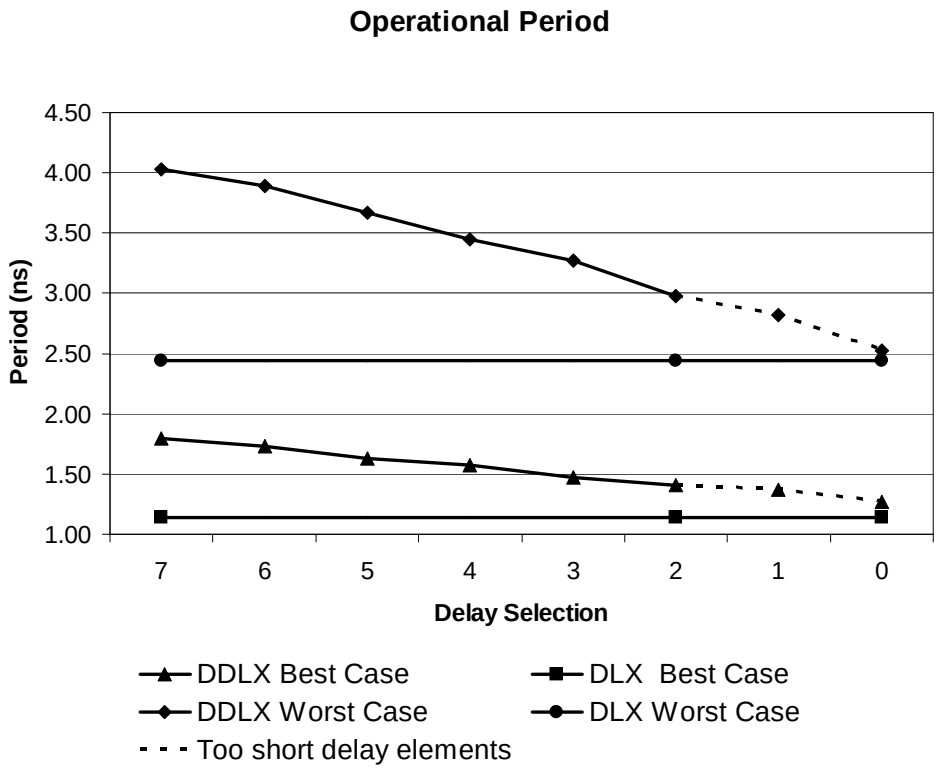


Figure 5.3: Timing results for DLX and DDLX.

is to operate the circuit at its real speed, not at their worst-case speed, *e.g.* under power supply, temperature or manufacturing variations. This means that on average a desynchronized circuit is much faster than the worst corner case. In Figure 5.4 the desynchronized real average case is compared to the synchronous worst case. We have assumed that the desynchronized real average case is a normal distribution between the two extreme cases, exactly like SSTA does for variability factors. The desynchronized circuit is presented to be faster than the synchronous one in 90% (the shaded area of the figure) of the cases. Moreover, the use of delay elements performs this frequency scaling dynamically and automatically without any user or designer intervention.

5.2.3 Power

The power consumption results were also taken by functional simulation. Initially, VCD (Vector Change Dump) files were written during the simulation and then the `Synopsys vcd2saif` tool was used to generate the equivalent SAIF files. Finally, the design was read along with the SAIF files in the `Synopsys Design Compiler` tool and the power reports were generated. Figure 5.5

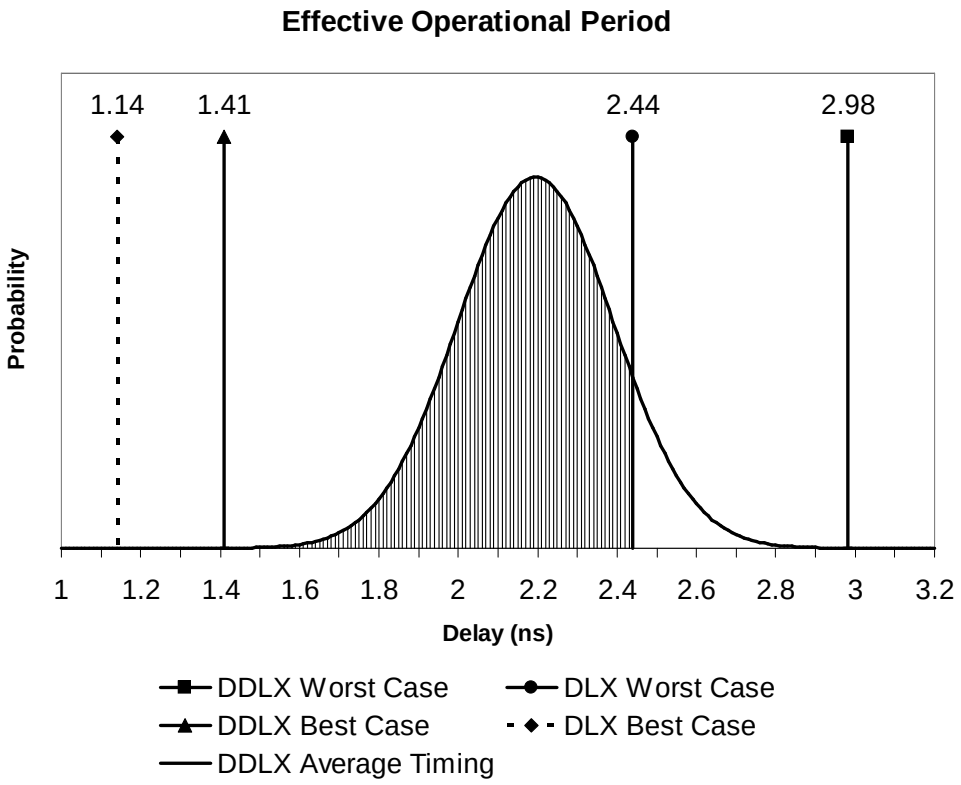


Figure 5.4: Real operation delay comparison between DLX and DDLX.

shows the comparative results of power consumption correspondingly to the previous ones of speed performance. The main overhead is again because of the increased number of cells due to the flip-flop substitution. The values are rising when the selection number lowers because the circuit operates in higher frequency.

5.3 ARM RISC CPU

The second design implemented is ARM966E-S microprocessor [20]. The synchronous design was already implemented and used as is for the comparisons with the desynchronized version. All the procedure followed is the same as for the DLX processor, except the fact that the Low-Leakage instead of the High-Speed version of the library is used, since the original synchronous circuit was implemented using this library.

Due to lack of any testbenches, only area results can be presented. Also, because of the ARM design's complexity and the limited knowledge of its inner architecture neither automatic nor manual grouping was possible. Thus, the ARM design was implemented using only one group. More complex

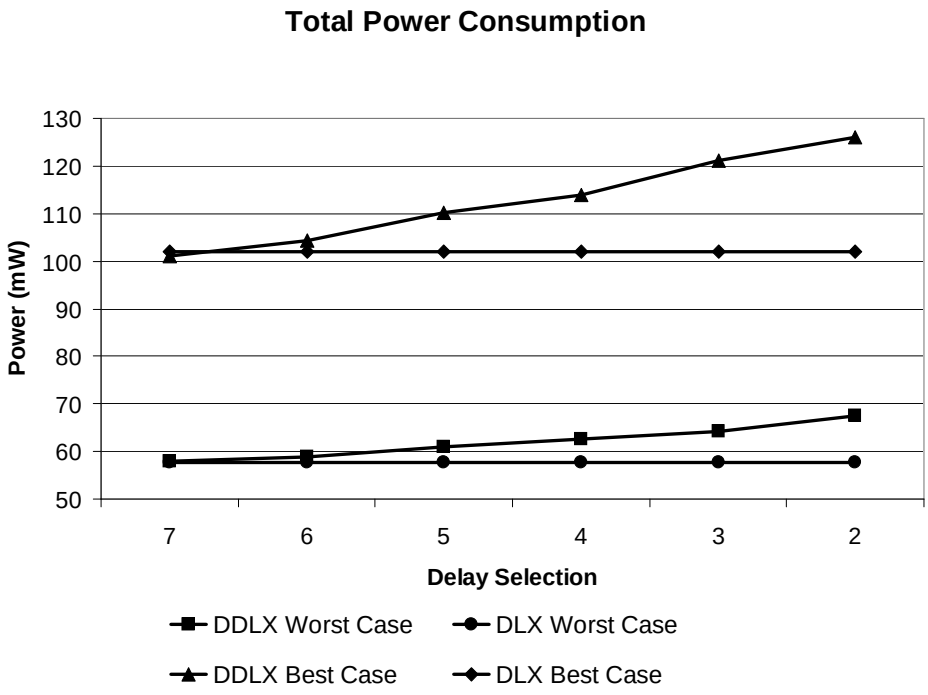


Figure 5.5: Power results for DLX and DDLX.

grouping algorithms supporting such complexity constitute part of the future work.

5.3.1 Area

The area results were taken from the Astro tool. The results in Table 5.2 are presented like the ones for the DLX. DARM refers to the desynchronized version of the ARM core. We notice that the results are similar to those of DLX and the total overhead (7.94%) is coming from the sequential elements. The ARM is a scan design. The combinational logic overhead because of the scan flip-flops substitution is included in the sequential logic overhead and this is why this overhead (40.70%) is significantly increased with respect to the corresponding one in the DLX design.

Design Phase	Area Property	ARM	DARM	
				% Overhead
Post Synthesis	# nets	34690	45626	31.52
	# cells	31549	45489	44.19
	cell area (μm^2)	578227.77	684791.86	18.43
	combinational logic (μm^2)	318108.19	318792.02	0.21
	sequential logic (μm^2)	260119.58	365999.84	40.70
Post Layout	# nets	38328	49514	29.18
	# cells	35218	49574	40.76
	standard cell area (μm^2)	633642.86	754822.12	19.12
	core size (μm^2)	792598.22	855551.00	**7.94**
	core utilization (%)	79.95	88.23	-10.36

Table 5.2: Area results for synchronous and desynchronized ARM.

6

Conclusions

This work has shown that the desynchronization methodology can be easily integrated to conventional industrial EDA flows without any major changes. The desynchronized circuits are implemented using the exact same procedure and tools, with the additional use of an automated desynchronization tool and the specification of some extra timing constraints. The results show that there is hardly any real overhead comparing to the corresponding synchronous versions and moreover, these results can be further improved by the existence of two-clock (master/slave) flip-flops in the target libraries.

This methodology allows the designer to have in an automated way all the advantages of a circuit without global clock. Firstly, the delay elements are influenced in the same way as the logic they match by both static (process) and dynamic (voltage, temperature) variability factors, at least for inter-chip variability. They are automatically and dynamically calibrated changing their delay value without the need of any extra circuitry or designer intervention, making this technique even better than binning. Moreover, voltage scaling is much easier since the delay elements are influenced in an analogous way as the logic.

The future work mainly consists of the implementation of more study case circuits to evaluate how much the results can be generalized. Moreover, SSTA can be used to verify how well the delay elements match the logic delay across the whole spectrum of operation conditions. Floorplanning constraints can be given to the backend tools to control the placement of the delay elements. Making the tools place them close to the logic they match more variability correlation is achieved. Also, after the final layout, Engineering Change Order (ECO) can be used to calibrate the length of the delay elements taking into consideration the final delays including full parasitics extraction. Finally, more advanced grouping algorithms capable of producing better grouping results for more complex circuits and multiple clock domain support can be implemented.

Bibliography

[1] S.R Nassif. Modeling and analysis of manufacturing variations. In *Proc. of Asia and South Pacific Design Automation Conference*, May 2001.

[2] Chirayu S. Amin, Noel Menezes, Kip Killpack, Florentin Dartu, Umakanta Choudhury, Nagib Hakim, and Yehea I. Ismail. Statistical static timing analysis: how simple can we get? In *DAC*, pages 652–657, 2005.

[3] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. A concurrent model for de-synchronization. In *Proc. International Workshop on Logic Synthesis*, pages 294–301, 2003.

[4] 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*, pages 149–158, 2004.

[5] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. From synchronous to asynchronous: An automatic approach. In *Proc. Design, Automation and Test in Europe (DATE)*, volume 2, pages 1368–1369, 2004.

[6] Abhijit Davare, Kelvin Lwin, Alex Kondratyev, and Alberto L. Sangiovanni-Vincentelli. The best of both worlds: the efficient asynchronous implementation of synchronous specifications. In *Proc. ACM/IEEE Design Automation Conference*, pages 588–591, 2004.

[7] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, April 2003.

[8] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.

[9] Evangelos Vlachos. Study of asynchronous controllers' circuits in de-synchronized systems. Technical Report 337, ICS-FORTH, 2004.

[10] D. Chinnery and K. Keutzer. Reducing the timing overhead. In *Closing the Gap between ASIC and Custom: Tools and Techniques for High-Performance ASIC design*, chapter 3. Kluwer Academic Publishers, 2002.

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

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

[13] J. Cortadella, A. Kondratyev, and C. P. Sotiriou. Coping with the variability of combinational logic delays. In *Proc. International Conf. Computer Design (ICCD)*, 2004.

[14] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. De-synchronization: synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design*, 0(0):0–0, January 2006.

[15] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[16] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[17] Stephen B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[18] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Transactions on Computers*, 44(11):1306–1317, November 1995.

[19] J. L. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publisher Inc., 1990.

[20] Advanced Risc Machines (ARM) Limited. *ARM966E-S (Rev 2) Technical Reference Manual*, February 2002.