HYBRID SYNCHRONOUS / ASYNCHRONOUS DESIGN

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Filipp A Akopyan

May 2011

HYBRID SYNCHRONOUS / ASYNCHRONOUS DESIGN

Filipp A Akopyan, Ph.D.

Cornell University 2011

In this new era of high-speed and low-power VLSI circuits, the question of which circuit family is best for a given application has become much more relevant. From a designer's point of view, process technology scaling continues to reveal undesirable device behavior. Thus, a designer has to make decisions not only at the micro-architectural scale, but also at a lower, circuit-level scale. However, common circuit families are not sufficient to solve modern engineering problems in many cases.

Our goal is to provide resources that will allow designers to select the circuit family that yields the best results in terms of power, area, and performance metrics for each application, with minimal human input. Using our techniques, this choice can be made in a timely manner without in-depth knowledge of each circuit family.

We propose an improved hybrid synchronous / asynchronous toolflow that offers significant reduction in the design cycle time and we advise on how our work can be extended to various types of circuit families for any given technology node.

We describe tools that we have developed to allow designers to implement their projects using both synchronous and asynchronous circuit families. We also present the cosimulation environment that we have developed to allow designers to run complex digital and analog simulations of various circuits at different levels of abstraction with minimal setup effort. Finally, we demonstrate a highly optimized synchronous-asynchronous interface that works as a bridge in designs where part of the logic is implemented asynchronously within a globally synchronous system.

Biographical Sketch

Filipp Akopyan was born in Moscow, Russia, where he grew up and lived until the age of 15. Filipp completed middle school in Russia and high school in the United States (at Spring Valley, NY). Filipp joined Rensselaer Polytechnic Institute (in Troy, NY) in September of 2001 and graduated number one in the School of Engineering with a Bachelor's Degree in Electrical Engineering in May of 2004. His concentrations at RPI included electronic circuit design and signal processing.

The author has been enrolled in a joint M.S. / Ph.D. program at Cornell University since September 2004. At Cornell his main interests included high-speed VLSI circuits (including 3-D integrated circuits and neuromorphic systems) that operate under extreme conditions and withstand process variations. He has also developed low-power asynchronous systems for signal processing.

Filipp is a part of the Asynchronous VLSI (AVLSI) research group led by Professor Rajit Manohar. Filipp's office is located in Upson Hall, 358 in the Computer Systems Laboratory.

"Desire to become the best at what I do drives my life."

dedicated to my family and friends; thanks for all the support

...yours truly

Acknowledgments

First of all I would like to thank my advisor, Professor Rajit Manohar, who always offered support and encouragement even in the toughest days of my graduate career. His enormous help, ideas and willingness to discuss my (at times risky and unconventional) ideas, allowed me to think outside of the box and perform the work outlined in this thesis. I thank my close friends and colleagues at AVLSI and the entire CSL staff. Special thanks go to Carlos Tadeo Ortega Otero, Ilya Ganusov, David Fang and Virantha Ekanayake for all the valuable advices that they have offered to me during my studies. I have been lucky to have such loyal and smart friends surround me in graduate school.

My appreciation and endless love go to my family (Andrey, Galina, Seva, Alexander and Vera), especially to my mother, Yuliya, who has always been there for me and gave up everything she could in order to make my life better. I would like to thank Sergey Rakov for all of his valuable professional and personal advices throughout my life and for getting me interested in the Electrical Engineering field.

My gratitude goes to my lifetime friends: Vadim Zipunnikov, Borjan Gagoski, Peter Paliwoda, Chin-Chen Lee and Jan Kostecki for helping me deal with difficulties of graduate life. I would also like to thank Alena for her love and patience through these last few years.

This research was supported in part by National Science Foundation and IBM.

Table of Contents

List of Figures

List of Abbreviations

ACT	Cornell's Asynchronous Circuit Toolkit
CHP	Communicating Hardware Processes, hardware description language
CMOS	Complementary Metal-Oxide Semiconductor
EDA	Electronic Design Automation
GND	Ground Power Supply Node
Netlist	circuit description at some level of abstraction
HSE	Handshaking Expansion
NFET	n-diffusion Field Effect Transistor
PFET	p-diffusion Field Effect Transistor
PRS	Production Rule Set, transistor pull-up and pull-down description
QDI	Quasi-Delay Insensitive
Sink	data token consumer
Source	data token generator
SPICE	transistor-level circuit description
TSMC	Taiwan Semiconductor Manufacturing Company
VDD	Positive Power Supply Node
Verilog	Hardware Description Language for electronic system modeling
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Motivation

Complexity of modern integrated circuits continues to increase with shrinking feature size. As a result of that, timing closure, power management and undesirable transistor behavior at high operating speeds have become increasingly important and challenging to deal with [34].

The main purpose of Computer-Aided Design (CAD) tools is to help designers combat these issues. Contemporary industrial design flow is well understood, documented, and constantly being updated and improved by large-scale CAD corporations. However, the limitations of current design flow are also well-understood [6], especially in the current era of low-power/high speed VLSI, where technology scaling leads to more parasitic analog effects that are only revealed during low-level simulations at the "end" of the design cycle.

Consequently, VLSI design is becoming more complex and more time consuming. One of the primary reasons for this is a lack of *fast and accurate* low-level circuit modeling tools. Presently, if a designer wants to perform gate-level opti-

mizations, he is forced to perform slow, transistor-level simulations that can take several days to complete for a reasonable size VLSI design. Furthermore, these simulations are often infeasible, since most current designs are standard-cell based and foundries do not reveal the exact structure of their gates to the designers due to confidential intellectual property agreements. Thus, designers are forced to perform most of their optimizations **only** at a high-level circuit description, where no notion of transistors or even gates exists. This type of decision-making process renders crucial low-level design space optimizations unavailable. In addition, mixed high- and low-level modeling is underdeveloped and cannot be performed with all known circuit families.

Moreover, if a designer attempts to perform low-level optimizations, he is required to have in-depth knowledge of circuit implementations and circuit families. It is extremely difficult to predict which circuit family will be ideal for a given design in a particular environment. Different optimization criteria exist for different applications, design constraints, and environments.

Ideally, a designer should have the freedom of describing a design using a high level language, such as Verilog or VHDL. The tools should then aid the designer with synthesizing this behavioral description into a lower-level description (possibly a gate-level netlist). Ideally, at that point the designer should already have some guidelines about which circuit families he should consider and which he should dismiss immediately due to the synthesized circuit structure. After a subset of circuit families is selected based on the guidelines, the full design or subset of the design (depending on size and complexity) can then be automatically synthesized into transistor-level netlists for selected types of circuit families, which can then be accurately simulated at the transistor-level. At this point, the designer should have enough data to make an educated decision as to which circuit family is best

suited for their design, given their set of requirements and guidelines.

1.2 Background

When selecting a circuit family, the designer has many choices. Synchronous families include static CMOS, domino logic, differential signaling, etc. However, many low-activity/high-speed applications may benefit from self-timed (asynchronous) circuit families, which offer tradeoffs in terms of throughput and/or power consumption in comparison to synchronous circuits [9]. Some potential benefits include data-driven switching activity and absence of clock circuitry, but these advantages come with the overhead of additional acknowledgement signals and potentially more complex data encoding (dual-rail signals, etc.).

Some application where asynchronous circuits have a clear advantage over conventional synchronous circuits are audio/speech processing [5] and neurobiological applications. A wide class of such designs has the property that the input signals don't change continuously; they are idle for some time, then change their value and return back to idle state. Consequently, a lot of power, if implemented in typical synchronous circuits, is wasted, because many samples of the input have the same values for a long period of time. For the purpose of eliminating the circuit power consumption when the input is stable, designers may use asynchronous design to implement signal processing operations. Asynchronous circuits are appealing for this task, since they don't have a global clock that forces periodic sampling. Unlike all the conventional synchronous circuits, an asynchronous system shuts down automatically, if the input is constant (i.e. the system is event-driven). In the applications of signal processing and neurobiological circuits, asynchronous circuits in many cases have several advantages in terms of energy consumption and complexity reductions.

Asynchronous circuits operate without a global clock, and use handshakes to transmit and control the flow of data as shown in Figure 1.1. The data-driven nature of asynchronous circuits allows a circuit to idle without switching activity when there is no work to be done. In addition, asynchronous circuits are capable of correct operation in the presence of continuous and dynamic changes in delays [24]. Sources of local delay variations may include temperature, supply voltage fluctuations, process variations, noise, radiation and other transient phenomena.

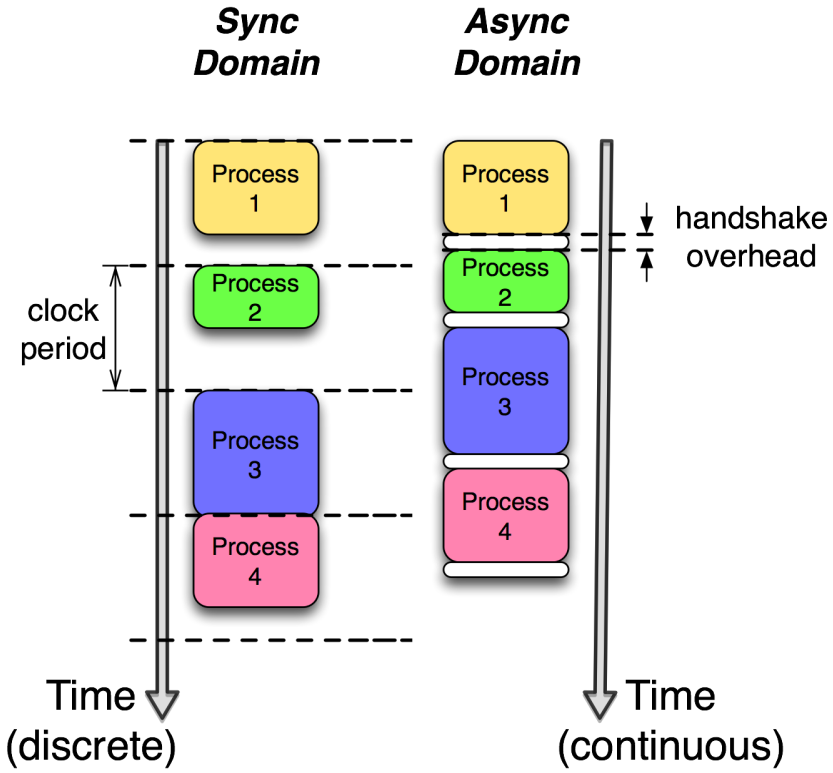

Figure 1.1: Synchronous and Asynchronous Time Domains

We believe that asynchronous circuits are a promising area in research because many design parameters can be improved, in comparison to the well-known synchronous circuits.

A brief comparison of synchronous and asynchronous circuit characteristics is presented in the table below.

	Synchronous Ckts	**Asynchronous Ckts**
Activity	Clock-Driven	Input-Driven
Power Consumption	Continuous	Activity-Dependent
Logical Correctness	Delay Sensitive	Delay Insensitive
Temperature Variation	Breaks Circuits	Immune
Radiation Immunity	Hard to Achieve	Achieved with Minor Modifications
Fault Tolerance	Requires Complex Circuits	Easily Achievable

1.3 Proposed Toolflow Basics

In order to allow the designer to make better-informed choices with regard to circuit families, we have developed a toolflow for automatic synthesis of a given logic block into synchronous and asynchronous logic families. This automatic synthesis enables a fair and systematic comparison between different circuit family implementations. After compilation of a given digital logic block into synchronous and asynchronous implementations, we can measure power, performance, and delay characteristics. Using our flow, the designer can evaluate various circuit types and quantitatively determine under which conditions an asynchronous circuit would result in reduced delay and/or power consumption compared to its synchronous counterpart or vice versa. In this fashion, the designer can quickly obtain the cost of each implementation in terms of power, performance, and area (transistor count), at which point he can decide which implementation should be used without going through the entire synthesis/layout of all blocks.

Our toolflow also provides highly optimized, pre-compiled cell libraries for dif-

ferent logic families, both synchronous and asynchronous, which eliminates the designer's requirement of thorough knowledge of all circuit families. All our tools are also compatible with industrial standard cell libraries. Such compatibility gives the designer another degree of freedom to pick the factory supplied standard cells if they are sufficient or beneficial for a given design, or to decide that another circuit family should be considered.

1.4 Asynchronous Design Methods and Challenges

For the majority of our asynchronous digital experiments, we have selected the quasi-delay-insensitive (QDI) style [24]. The advantage of this family is that the resulting circuits are robust and insensitive to process, voltage, and delay variations. Electromagnetic emissions are also minimized as compared to synchronous circuits due to the absence of a high-frequency clock signal throughout the circuit. A sample conventional asynchronous compilation method is described in Alain Martin's communicating process compilation technique [23]. It is based on a synthesis method that translates a high-level design description to circuits through handshaking expansions, and production rules. This technique facilitates physical circuit realization, where some of the decomposition in Martin's approach can be generated automatically.

The issue with all the current asynchronous toolflows is that they are not easily combined with their synchronous counterparts. One cannot easily interchange synchronous and asynchronous circuits within a design. *No* unified tools that could handle both circuit families exist according to the author's knowledge. In general, the lack of EDA (Electronic Design Automation) tools for asynchronous flows limits their usage in contemporary electronic systems. Most of the synthesis and simulation of asynchronous circuits is presently performed by hand, which puts

a large burden on the designer and requires in depth knowledge of the asynchronous circuit theory. In addition, methods of combining synchronous and asynchronous circuits are limited and sometimes do not satisfy the requirements of the more complex systems.

Our goal is to solve these problems and introduce a new hybrid synchronous / asynchronous toolflow that allow designers to easily build not only synchronous, but also asynchronous systems and to have an efficient way of combining these two circuit families.

1.5 Toolflow Evaluation

Transistor models will inevitably deviate from the actual manufactured integrated circuits. All the results presented in this thesis are based on the theoretical models, simulations and manufacturing data from previous test-chips/test-runs. However, QDI asynchronous circuits tolerate variations between models and physical devices due to the conservative design methodology [20]. Conversely, for some aggressive synchronous circuits, more timing-closure analysis should be performed by designers using the parasitic extractions obtained after the complete place-and-route step, which is well covered in the industrial flows and is not presented in this work. For the majority of measurements and tool calibration presented in this thesis we have used the Nangate 45nm Open Cell Process/Libraries [1], commonly used for testing and exploring modern circuits and EDA flows.

For the toolflow evaluation purposes we have designed various prototypes of integrated circuits. While developing the presented toolflow, we utilized many of our own tools, described in this thesis; as well as several industrial tool packages. The following industrial tools have been extensively studied and used in our experiments; for layout: Cadence IC 6.13, Cadence Encounter, Micromagic MAX;

for high-level simulations: Synopsys VCS, Synopsys Design Compiler; for analog simulations: Synopsys HSIM, Synopsys Hspice. These software tools allow us to perform pre-layout simulations, Verilog- and VHDL- type synthesis, post-layout analysis, and verification of different digital and analog circuits.

1.6 Organization of This Thesis

This thesis is divided into four major parts.

In the first part we will discuss our innovations in toolflow development. We will demonstrate a hybrid synchronous/asynchronous toolflow that uses identical mechanisms for synthesizing a given design into both synchronous and asynchronous circuits. This toolflow gives designers the flexibility of obtaining accurate power/throughput/area estimates early in the design cycle using different circuit families.

Second part of this thesis will focus on the COSIMulation tool that we have developed in order to allow designers to simulate their designs at various levels of abstraction. This tool unifies various industrial and our own digital and analog simulators to give designers an opportunity to test their designs. This tool handles both synchronous and asynchronous circuit families and has the ability of automatically connecting different types of input and output environments to the design to provide logical and timing correctness checks.

Third part demonstrates the effectiveness of our toolflow. We evaluate our gate-level conversion methods on a set of ITC-99 benchmarks, compiling them into a highly efficient static CMOS industrial library and into our own QDI asynchronous library. We present throughput and power consumption trade-offs in relation to circuit input signal activity. We also perform a design space study based on power consumption, input activity factor and maximum supported frequency.

The last part focuses on a highly efficient asynchronous-synchronous interface. This interface is most useful in globally synchronous systems, where part of the computation needs to be implemented asynchronously due to some targeted metric or design characteristic. Our interface guarantees deterministic timing of the overall system due to the implemented synchronization protocol.

Chapter 2

Toolflow

2.1 Toolflow Comparison

2.1.1 Conventional Toolflows

The toolflow used by the majority of companies in industry is presented in Figure 2.1. First, the designer creates a high-level Verilog/VHDL description of the circuits. Second, this description is synthesized into a gate-level netlist using a specific set of standard cells with pre-layout timing estimates supplied by the foundry. Third, the designer works with automatic place and route tools to obtain a physical (layout) implementation of the circuit – in practice, this step is not fully automatic and requires a significant amount of manual effort. At this point the layout can be extracted, but the contents of the standard cells are not revealed by the foundry. Only after these steps are completed can the designer perform **analog-level** simulations with the estimated parasitic elements from the layout – for the first time since the beginning of the design cycle.

The place and route step takes a **large** portion of the design cycle time and

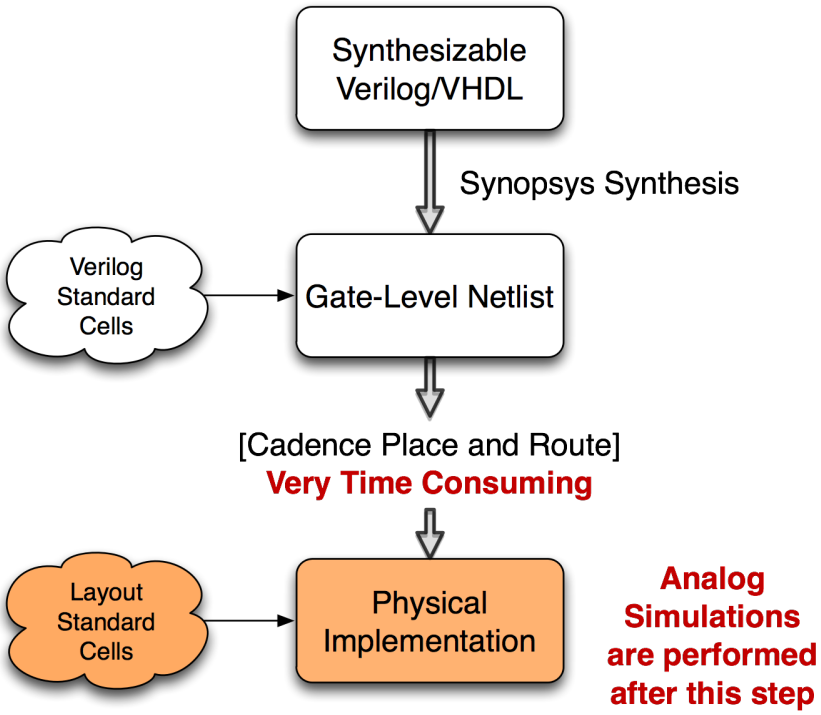

Figure 2.1: Industrial Toolflow

needs to be partially or completely repeated after every modification to the design. It also takes the designer a long time to get to the first set of analog-level simulations, where many common problems, such as cross coupling, charge sharing, signal swing issues are revealed.

As for the asynchronous toolflow, different design teams around the world have their own toolflows. Various tools have been developed to support asynchronous circuits, however, a large portion of the design process still requires designer's intervention for synthesis and analysis. A toolflow used by Cornell's AVLSI research group is presented in Figure 2.2.

Some of the tools shown in Figure 2.2 will be briefly described later in this thesis. From the comparison of the two flows, it is obvious that synchronous and asynchronous toolflows use completely different description languages and tools. No parts of these flows may be interchanged and, thus, evaluation of synchronous

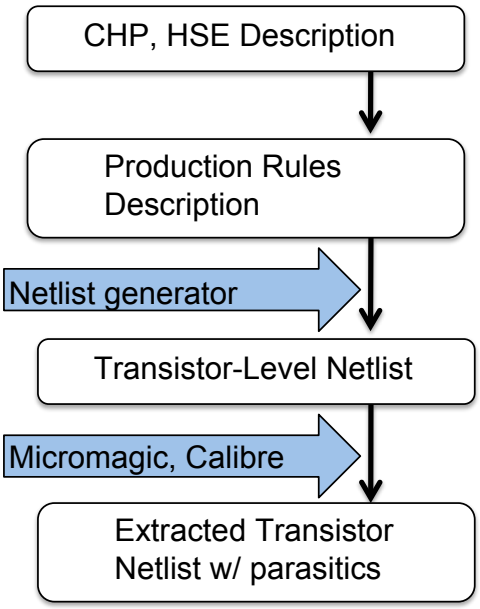

Figure 2.2: Asynchronous Toolflow

and asynchronous circuits conventionally requires a lot of work and a full understanding of asynchronous circuit operation to perform some of the manual synthesis.

2.1.2 Proposed Toolflow

Our proposed toolflow, shown in Figure 2.3, provides full support of both synchronous and asynchronous circuit families. It uses same exact tools to generate both synchronous and asynchronous netlists. It also allows easy modifications/adjustments for parts of the design. This toolflow eliminates all the manual labor previously required from the designer to perform asynchronous circuit synthesis.

Our toolflow also removes the necessity for the place and route step for all the preliminary design decisions and measurements. Once the design is finalized and satisfies all the metrics, the place and route step is performed only once with some

minor post-layout adjustments to account for cross-talk, transmission line effects, etc.

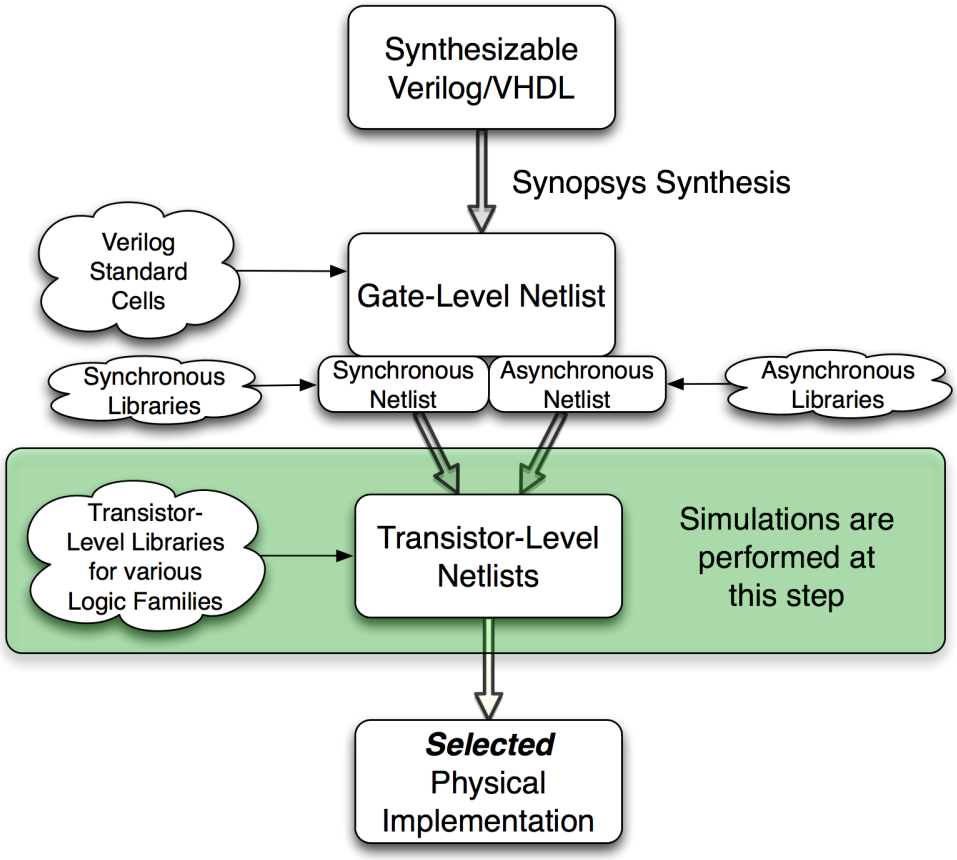

Figure 2.3: Proposed Toolflow

In order to minimize design time and allow the designer to test various types of circuit families for a given design, we augment the industrial toolflow in the following manner. Similar to the current industrial flow, the designer creates a high-level Verilog/VHDL description of the circuits, which is then synthesized into a gate-level netlist using a specific set of supplied standard cells. At this point the designer, instead of using the "black-box" industrial cells, has a choice of using cells from various different circuit family cell libraries. For example, if the selected circuit family is synchronous, the synthesized netlist is used directly with transistor-level libraries of various synchronous families. However, if the selected

family is asynchronous or data-driven, we perform several netlist transformations, described in the next section, to obtain a logically equivalent asynchronous gate-level netlist. In this case, asynchronous libraries are attached to the obtained gate-level netlist. Presently, our tools perform the transformation of synchronous gate-level netlists into Quasi-Delay Insensitive (QDI) [24] asynchronous netlists, but it is relatively simple to perform a similar set of transformations to obtain a bundled-data asynchronous netlist, for example. The designer need **not** have an in-depth understanding of the operation of asynchronous circuits, because our tools automatically perform correct netlist transformations and attach the corresponding libraries of asynchronous components.

The advantage of our flow is that, at this early point in the design cycle, designers can perform analog simulations using industrial simulators that take into consideration most of the parasitic effects of the given design. To enable this, we estimate and annotate wiring capacitance associated with each gate's output node. Once the designer is satisfied with the simulation results, the physical implementation or layout step is performed only once using the circuit family that gave the best results for the specified metrics.

2.1.3 Proposed Simulator Chain

In the toolflow that we have presented, the designer has much more flexibility simulating the circuit at various pre-layout levels of development, as shown in Figure 2.4. As in the industrial flow, the Verilog/ VHDL behavioral code may be simulated with an industrial level simulator, such as Synopsys VCS [4]. After compilation into gate-level description, the synchronous gate-level netlist may be simulated with the same high-level simulator, whereas in the asynchronous case we use our digital simulator PRSIM, which we describe in the next section. We

have also developed an interface to allow the cosimulation of synchronous and asynchronous circuits simultaneously at various levels of abstraction.

After our netlist transformations are performed, from a gate-level description we produce a transistor-level synchronous or asynchronous netlist that can be simulated using any of the industrial analog simulators, prior to the layout step. After the physical implementation is performed, the same analog simulator may be used again for final evaluation of the design.

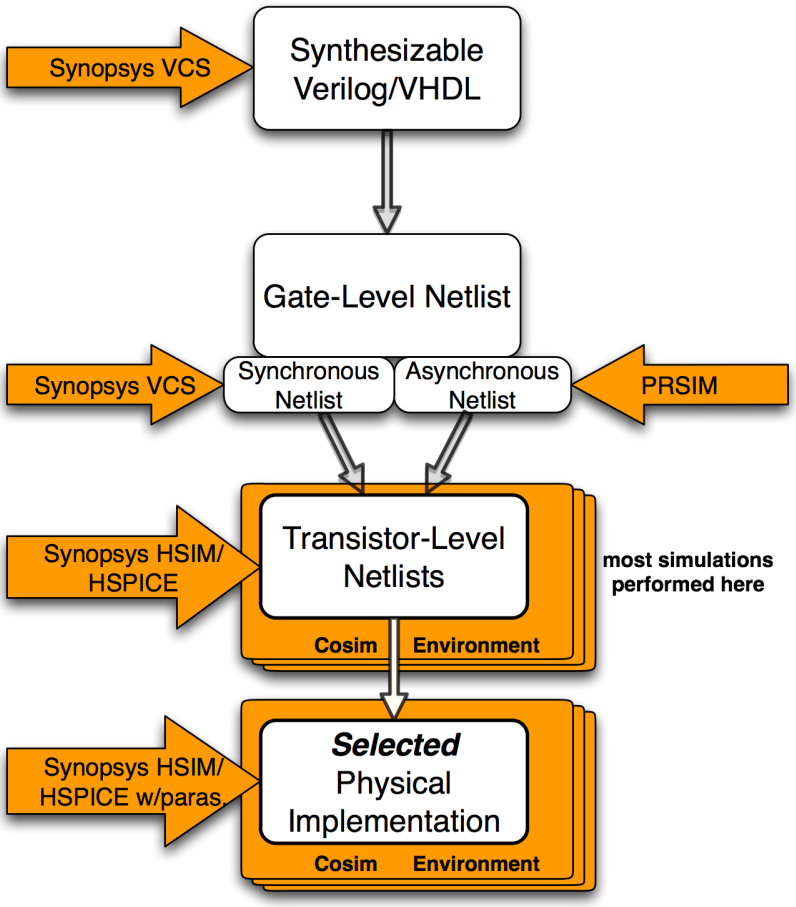

Figure 2.4: Sample Simulator Chain

2.2 Industrial Tools Used in the Flow

2.2.1 Synchronous Digital Simulator

For all behavioral level simulations, as well as for gate-level synchronous netlist simulations we have used Synopsys VCS. The advantage of such a simulator is that we can compare results obtained from circuit descriptions before and after we invoke Synopsys's synthesis tools.

2.2.2 Transistor-Level Analog Simulator

For all of our transistor-level simulations we have used Synopsys HSIM for larger circuits and Synopsys HSPICE for more accurate smaller circuit simulations. Our tools generate netlists that account for gate and parasitic output capacitances, as well as estimated average-case wiring capacitance to accurately represent delay and switching energy.

2.3 Cornell AVLSI's Tools Overview

2.3.1 Asynchronous Digital Simulator (PRSIM)

The simulator we use for our experiments with asynchronous circuits is an event-driven digital simulator. It has been extended to evaluate the transient effects of temperature and supply voltage on delay if so desired. The input to the simulator is a simplified transistor-level netlist (automatically generated) in the form of event rules describing the logic. PRSIM has the advantage of performing fast digital simulations, while simultaneously testing for correct asynchronous circuit behavior.

2.3.2 Verilog-to-ACT

The Verilog-to-ACT tool converts a synthesized Verilog netlist into an equivalent intermediate format gate-level netlist that hierarchically describes pull-up and pull-down transistor stacks of each gate used in the design. This tool can be used to automatically generate either a synchronous netlist or an asynchronous netlist.

The transformation from a Verilog gate-level netlist to synchronous transistor stack-level netlist is straightforward. Only the semantics of the netlist are altered to transfer Verilog-level description to an intermediate ACT description. At this stage, an appropriate transistor-level library is attached to the circuit description—various synchronous families may be used.

If an asynchronous QDI netlist is selected, the tool produces an asynchronous gate-level netlist, by performing gate level transformations based on the initial synchronous gate-level netlist. Asynchrony is completely transparent to all the previous parts of the toolflow, since prior to this step the circuit looks and behaves completely synchronous. This kind of transformation preserves the circuit structure. It replaces synchronous gates with their asynchronous counterparts, as well as inserts asynchronous-specific circuit modules, as described further.

Verilog-to-ACT converts every boolean channel into a delay insensitive channel and automatically inserts copy processes for high fan-out signals and token sources for constant-value inputs. In addition, during the asynchronous transformation, all flip-flops are replaced with asynchronous initial token buffers. The performed transformations are outlined in Figure 2.5.

An example of a Verilog-to-ACT transformation, where the asynchronous netlist is produced from a synchronous netlist is demonstrated in Figure 2.6. In this figure, a synchronous netlists consist of a toggle flip-flop (T flip-flop), which is implemented by using a D-type flip-flop with it's output going to an XOR gate, along

Synchronous Netlist	Asynchronous Netlist
Booleans	Asynchronous Channels
Synchronous CMOS gates	Asynchronous CMOS Gates
VDD and GND inputs	1- and 0- Data Sources
Dangling nodes	Data Consumers
Gate Output Fanout	Copy Process
Synchronous Flip-Flops	Initial Token Buffers

Figure 2.5: Verilog-to-ACT Conversion Table

with the T flip-flop's input; the output of the XOR gate is connected to the input of the D flip-flop. This is a standard way of implementing a toggle flip-flop. After the synchronous-to-asynchronous transformation takes place, the asynchronous netlist looks like the right part of Figure 2.6. D flip-flop is substituted with an initial token buffer, fan-out of two is replaced by a two-way copy process, and the XOR gate is implemented in an asynchronous manner.

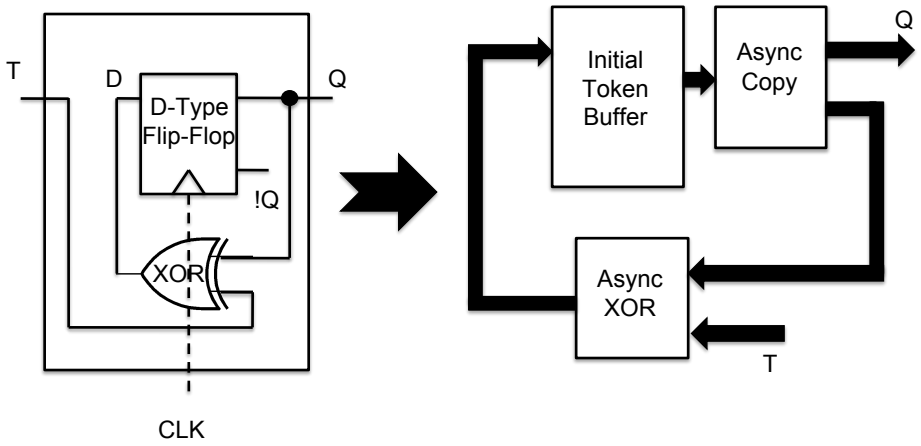

Figure 2.6: Verilog-to-ACT Synchronous to Asynchronous Transformation

After the transformation is completed, the obtained asynchronous circuit emulates synchronous circuit's behavior. All the clock actions are replaced with asynchronous handshakes, but the sequencing and structure of event occurrences in the

system remain untouched.

These types of transformations are suitable for small and medium sized circuits. This technique is not as efficient for large scale designs, since many asynchronous-specific design optimizations are not performed due to preservation of the synchronous circuit structure.

Eventually, for more efficient conversion, we would like to perform the synchronous-to-asynchronous transformation at a higher-level behavioral description, but that would require a complex compiler. Therefore, at this time we have decided to stay with the gate-level transformations. Once the high level transformation is implemented, we can use methods such as automated concurrent pipeline synthesis, as described by Teifel [35], to perform the asynchronous synthesis function.

2.3.3 Netlist Generator

The Netlist Generator tool is used to automatically generate a hierarchical spice netlist from a previously described ACT netlist, generated by our Verilog-to-ACT tool (the pull-up and pull-down transistor description for a given circuit family's library). Values that control process-specific parameters such as gate input and output capacitances, approximated wiring loads, minimum p- and n- transistor sizing, source/drain area and perimeter, and spacing between two FET-s in the same diffusion stack are specified to allow the Netlist Generator to calculate parasitic capacitances.

2.3.4 Automatic Cosimulation Environment Generator

We have created Automatic Cosimulation Environment Generator (COSIM) to allow co-simulations of an arbitrary mix of synchronous and asynchronous circuit-families at various levels of abstraction. The environment generator automatically

creates an interface between the Verilog simulator VCS, the PRSIM asynchronous simulator, and the HSIM transistor-level simulator. This tool allows the designer to test mixed synchronous-asynchronous circuits while implementing the actual circuits using different families and different levels of abstraction. One can cosimulate a high-level Verilog/VHDL synchronous circuit description together with an asynchronous pull-up/pull-down circuit description, as well as with a transistor-level description that includes all the transistor parasitic effects and wire-load estimates. The tool can also automatically create an appropriate test-bench for a given circuit with both input and output environments. COSIM will be described in more detail in Section 3.

2.3.5 Circuit Family Libraries

We have created transistor-level libraries for various circuit families that give us the capability to synthesize our circuits into their corresponding transistor-level netlists using different synchronous and asynchronous families. Post synthesis (pre-layout) analysis allows us to pick the best circuit family depending on a targeted metric such as power consumption, area, throughput, latency, etc.

Delays and capacitances are based on the logical effort model and are calibrated against a realistic technology node in all our simulations. Please note that it is straightforward to calibrate the libraries for a different/new technology node. The only parameters that are changed with a new process are the circuit properties and transistor descriptions – minimum size, parasitic parameters, mobility values, etc. – used for the Netlist Generator configuration.

For the asynchronous library style we have implemented the QDI circuit family [18] due to its robustness to delay, temperature, and process variations as described in the Introduction section of this paper. We have implemented our standard cells

using PCEHB type handshake reshufflings [16]. Other asynchronous logic families such as bundled-data may be used instead with appropriate netlist transformations.

Chapter 3

Automatic Cosimulation Environment Generator

3.1 Motivation

The main reason for creation of an Automatic Cosimulation Environment Generator (COSIM) is current designer's inability to perform mixed-level synchronous and asynchronous circuit simulation at various levels of abstraction. According to the knowledge of the authors, there are presently no tools that can automatically perform such synchronous/asynchronous circuit simulations, along with proper modeling of the interfaces between the two domains.

With the complexity of current chip design increasing drastically, it is becoming impossible to perform transistor-level simulations of the entire chip. We would like to have an option of performing partially transistor-level and partially high-level behavioral simulations of the circuits.

As for simulation of purely asynchronous or mixed synchronous-asynchronous

systems, another problematic part of the simulation is creating the correct testing environment. In order for an asynchronous environment to function correctly, it has to support a number of different types of channels, have the ability to properly assert data validity and enable signals in order to support various types of asynchronous handshakes. Creating a proper environment is not a trivial task. Implementing such environment at transistor-level becomes very complex and, often, the obtained environment doesn't check all the possible scenarios that can lead to a circuit malfunction. Thus, we would like to implement this environment at a higher, behavioral description level. This type of environment must not interfere with circuit operation, but at the same time must perform extensive circuit testing using various input signal patterns and using different timing situations. Also if we have such environment written at higher level of description, it will be very easy to isolate this environment into a separate circuit domain so that all our throughput, area and power measurements are not influenced by the environmental artifacts that should not be accounted for during measurements.

Presently Synopsys tools, such as HSIM and Nanosim offer some "hooks" to perform cosimulations of behavioral-level and transistor-level descriptions. Unfortunately, these hooks require a lot of manual work to setup such a cosimulation. This work has to be repeated for each circuit that is being tested and it requires thorough understanding of the circuit's operation to setup all the input and output streams correctly. Also, Synopsys provides no support for testing asynchronous circuits or mixed synchronous /asynchronous circuits, which is the major focus of our work.

As described later in this section, we invoke several modern digital and analog simulators (some of these are simulators created by the AVLSI group here at Cornell). We offer designers to invoke their circuits at various levels of abstraction,

which allows trading higher accuracy of simulation for shorter simulation time and vise versa.

In the work outlined in this thesis, we go further than just cosimulations of different circuit families, but we also integrate all the synthesis and measurement tools that provide a full package in synchronous/asynchronous circuit testing and analysis. The following industrial and Cornell's tools were utilized in COSIM's development and are briefly explained in this thesis: Synopsys HSIM, Synopsys VCS, Cornell's PRSIM, Cornell's TLINT, Cornell's Netgen.

All the COSIM development illustrated in this section has been done in collaboration with my colleague ***Carlos Tadeo Ortega Otero***, who is also a Ph.D. student in Cornell's AVLSI research group.

3.2 Vision

Ideally, we would like to have a unified interface that could be used by designers for simulation run control, as well as all pre- and post- circuit processing, such as compilation into a spice-type netlist, power measurements, area estimates, throughput measurements. We would also like to have minimal amount of setup performed by the designer, but at the same time we don't want to take away all the functionality provided by the tools. So in most cases we create a cosimulation environment with a default set of parameters that provide a good simulation time / accuracy tradeoff. The designer may easily modify these parameters without going back to any of the Verilog/VHDL/SPICE files. All the parameters for all tools may be specified in the top-level simulation setup file, the only file that the designer has to create in order to invoke all the tools that we use. The designer never specifies any simulator setup or post-processing commands; he only specifies the necessary simulation parameters and the circuit nodes he would like to monitor. COSIM

automatically invokes he correct sequence of commands to run the appropriate tools in the correct sequence.

In order to connect Cornell's and Synopsys's simulators we use a VPI (Verilog Procedural Interface) created here at Cornell by our advisor. Synopsys's VCS natively supports VPI-type calls, which allows us to connect our own event-rule digital simulator PRSIM, VCS and HSIM. We can then guarantee proper interaction between simulators that are working with different levels of circuit abstraction. VPI for connecting VCS and HSIM is provided by Synopsys, but requires quite an amount of manual setup and compilation; we take that burden off the designer and perform all the setup automatically as well.

As mentioned previously, we would like COSIM to generate all input pattern generators (called *sources*) and output interfaces (called *sinks* or *buckets*). For source, the only information the designer has to provide to COSIM is an input stream file in text format, or to pick the type of input signal probability distribution he would like to use. COSIM automatically detects the types of channel that it is dealing with. It then connects appropriate sources and sinks. Sinks in COSIM can perform several functions: they can simply monitor the output; they can compare the outputted values with a set of predefined values; they also can perform handshakes at the output in case of QDI asynchronous circuits. All of this functionality is built into COSIM and the designer has to only select what type of buckets to use.

3.3 Auto Simulation Setup

In the process of creating COSIM, one of the major tradeoffs that we were facing was the amount of manual control that should be given to the designer during the simulations setup. More control requires more in-depth understanding of the

circuit's operation, as well as better understanding of the *correct* environment behavior.

We decided to require the designer to perform as little setup as possible, with COSIM making the educated guesses as to circuit's behavior and it's interfaces with the environment. For example we have an option of automatically adding source and sinks to all the circuit's inputs and outputs. COSIM decides how to reset these sinks and sources, and what type of input pattern distribution to pick. However, the designer always has an option of specifying as much detail as he would like (for example for all the critical signals), and afterwards COSIM can perform the rest of the setup automatically.

This approach allows the designer not to have in-depth understanding of all the tools that are used in the simulation process (though such "ignorance" is not recommended). If auto-connection is used at inputs and outputs, the designer does not have to be familiar with the exact handshake reshufflings (though if non-standard reshuffling is used in the circuit, this may lead to a deadlock during the simulation), pattern setup, etc.

Majority of the simulation parameters have default values that are assigned to them. The designer may override all of these values in the unified setup file. The default values were chosen to provide, in our opinion, a reasonable degree of simulation accuracy while still keeping a practical simulation time.

COSIM automatically performs various types of checks during the simulation run. These include node capacitance calculation check, periodic excessive current check (test for shorts), dangling nodes check (nodes that are not driven), unconnected node check, non-switching node check (except global signals like power and ground), power high-impedance node check, rise/fall time check, etc.

All the feedback from COSIM and from the simulators is collected and reported

back to the designer through various report files. COSIM warns the designer about potential errors in the designer's setup; it also informs the designer about the setup that it performed automatically to make sure that it didn't make any assumptions that are not consistent with the designer's implications.

3.4 Overview

COSIM is a PERL-based program that functions in the following manner. A designer creates only one unified setup file in the XML format. COSIM parses that file and sets up all the necessary simulators, interfaces between the simulators and prepares all the measurement and post-processing tools.

COSIM passes the main simulation directives to the VCS simulator that monitors the run at the top level of abstraction. COSIM creates the top-level Verilog-based wrapper that has the global parameters, such as simulation time, circuit module, global variables, all the necessary wiring connections, signal names, top-level delays, timescale, top-level circuit inputs and outputs, etc. The created Verilog wrapper also connects the selected Verilog environment (sources and sinks) to the design. Connections to PRSIM signals for asynchronous digital circuits described in ACT are also performed here.

COSIM also creates a spice wrapper, if this option is selected, for simulating transistor level netlists (including extracted circuits). The spice wrapper specifies all the necessary low level details and parameters to correctly invoke HSIM simulator. Among these parameters are supply voltage, output file format, simulator accuracy parameters, Verilog rise and fall times for interface signals, analog iteration method for initial operating point calculation. A table with some of the variables that may be optionally set in the unified interface file (and will appear in the spice wrapper) is presented later in this chapter of the thesis. Spice wrapper

also sets up all the analog voltage and current sources, global variables, connections with the transistor-level circuits, all the measurement commands, etc.

Besides the main two wrapper files for VCS and for HSIM, COSIM automatically creates various other files that are transparent to the designer under most circumstances. If designer selects an ACT-type file and specifies that he wants to simulate the circuit at transistor level, i.e. by invoking HSIM simulator; COSIM automatically runs Netgen, and generates a top-level spice, as well as all the spice subcircuit instances that were used in the original ACT description. Similar to the command-line Netgen run, COSIM reads in a Netgen configuration file, which should be either in local directory where COSIM is run, or specified as a *netgen_cf* variable in the unified interface XML file.

We use a simple JAVA-based parser to determine the names of the circuit ports in all *defproc-s* of the ACT file in order to correctly connect these ports in the Verilog wrapper and the spice wrapper. If the designer uses "!" and "?" to indicate directionality in the top-level *defproc* of the ACT file, we also parse this information and later use it if the designer wants to automatically connect sources and sinks to the given module. In that case directionality only has to be indicated at the top level module. Besides determining the names and directionalities of the ports, we also use the parser to determine the types of channels that we are dealing with. With that information, the user doesn't need to specify channel type in the XML file, since this information is obtained automatically.

When running Netgen directly from COSIM, several additional files are produced, that do not appear during a command-line Netgen run. These files have to do with connection of ACT-type modules to VCS. COSIM produces a file that contains the parsed channel types and names (as mentioned previously), located in the *.act.AUX_PROC_PARSE. It also produces a file with top level Verilog mod-

ule connections that specify input, output types for the Verilog wrapper, located in *.act.v. We also produce a *.act.v.port_info file which contains all the channel name equivalences between Verilog and ACT. Verilog by default has no notion of channels, it only works with wires; we thus have to mangle some names for Verilog, since it does not accept some of the characters used in ACT naming. These files are transparent to the designer and will not be described in further detail here.

When setting up the VCS-HSIM mixed signal simulation, COSIM creates a file call cosim.cfg, which holds several HSIM command line arguments that are used in the cosimulation initialization. Specifically, the file contains the names of all the modules that need to be simulated in the analog (transistor-level) mode. This file also sets up several initialization arguments, like the name of the spice wrapper. This file is also transparent to the designer.

Another important file that COSIM creates automatically is nsda_cosim.sp. This file is passed during the cosimulation initialization to the HSIM analog simulator. This file contains the top level spice module with all the port names that correspond to the appropriate names of the Verilog-type environment (sources and sinks). The main purpose of this file is to indicate to the simulators the VCS-to-HSIM and HSIM-to-VCS interface signals. PRISM connections are treated similar to VCS connections; signals from PRSIM first go to VCS and then to HSIM, if desired. Once the interface signals are identified the simulator automatically performs analog-to digital (HSIM-to-VCS) and digital-to-analog (VCS-to-HSIM) conversion of the appropriate signals to guarantee correct communication between the digital and analog simulators. nsda_cosim.sp is essential for passing values back and forth between the two simulators. This file is also transparent to the designer.

While setting up the proper environment for the circuit that is being tested, COSIM parses all the interface channels specified by the designer in the unified

interface XML file and automatically determines the channel types from the ACT file (if the top level module was described in ACT-format), as stated earlier. If some of the ports of the top-level module are not specified in the XML file, COSIM will notify the user that there are dangling ports in the module. For the input channels, the user has an option of either passing a file with a given input stream, or selecting an input probability distribution. If nothing is selected, COSIM will use a uniform input probability distribution for the channel by default. As for the output, the user specifies whether he wants to: simply watch the values of the channel, record the values, perform handshake, and/or compare the values to a predefined output pattern. By default, COSIM will simply watch and record the outputted values on every channel.

Once all the circuit interface channels have been identified, COSIM creates an environment_used.v file which contains all the proper sources and sinks along with their options for all the input and output channels (including CLOCK source, if there is one). COSIM looks up the required sources and sinks in the environment library that we have created. This library contains various flavors of the environment elements. The description of our library will be provided in a later section. environment_used.v contains only unique instances of each type of sinks and sources used. This file is created automatically and passed to the simulators during the VCS-HSIM cosimulation initialization.

3.5 Functionality

The main purpose of COSIM is to minimize the effort required by the designer to simulate his design using various levels of hardware abstraction, and using different circuit families, including mixed synchronous/asynchronous circuits. COSIM simplifies this process and allows the designer to perform circuit simulation with only

rough understanding of all the simulators and circuit implementations. COSIM also performs a task of automatically creating a testing environment for the design.

As mentioned previously, the current COSIM implementation requires the designer to create only one top-level unified interface file in the XML format. I shall refer to this file as *cosim.xml*. This file allows designers to perform connections of various circuit parts implemented in different description languages and at different levels of abstraction. An example of such a design is represented in Figure 3.1.

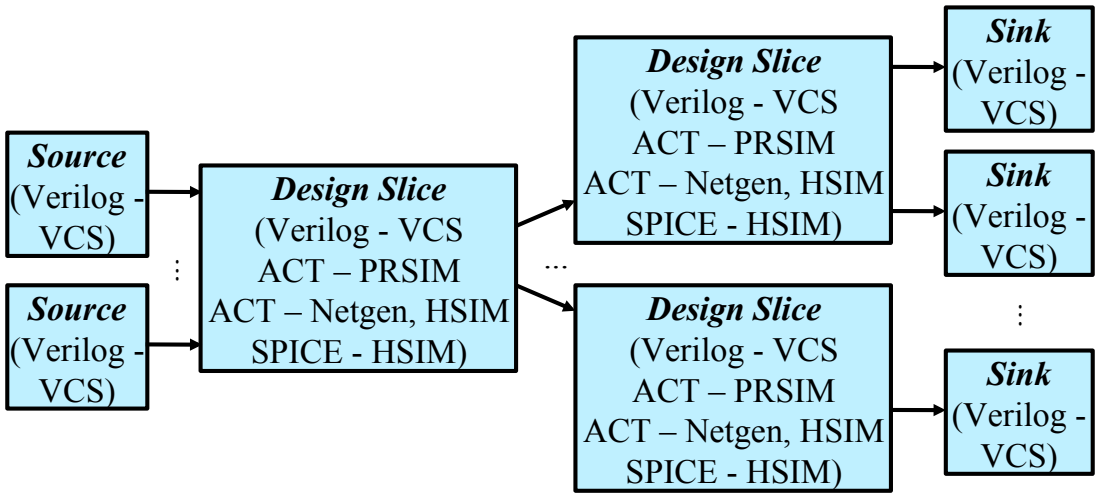

Figure 3.1: Overview of COSIM's Functionality

The testing environment (sources and sinks) for all the designs is created in Verilog, as shown in Figure 3.1. This allows us to perform more complex input/output value comparisons, logical correctness and timing correctness checking. In Verilog we can easily implement various functions that help designers debug their circuits.

All values originate in VCS and are then passed to PRSIM or HSIM; as for the outputs, all the values travel back to VCS, where various checks are performed (based on designer's selection in cosim.xml) to determine whether the circuit is functioning properly.

As for the actual circuit implementation (outside of the testing environment),

labeled as *Design Slice*-s in the above figure, they may be implemented in Verilog, ACT or in spice formats. If the circuit part is implemented in ACT, the designer has an option of either using PRSIM for simulation, or, alternatively, of running Netgen and generating a transistor-level description of his circuit and then simulating it using HSIM (assuming all the production rules are properly sized by the designer).

These *Design Slice*-s may be connected in cosim.xml in any arbitrary sequence, as depicted. Sequential and parallel compositions are both allowed. Any organization of Verilog, ACT and spice modules is allowed. The designer also has the option of implementing sources and sinks internally in his circuit, however, most of the time that would limit the testability of the circuit, which is why we advice designers to use our *smart* Verilog environment.

If an improper connection is performed in cosim.xml, or a violation occurs in one of the sinks or sources (timing violation, value mismatch during comparison, etc.), COSIM will inform the designer of this violation and may terminate the simulation if desired by the user.

For each top-level module (most of the time just one), the designer needs to specify some information in cosim.xml. Specifically, the designer indicates:

$< circuit\ sim\_type = "spice"\ file = "prs/buffer.act"\ instance = "pcfb\_buf"$
$file\_type = "act"\ name = "n1" >$

...

$< /circuit >$

The description of all the parameters for the circuit module is provided in the following table. Source and sinks should also be included within the above *circuit* description block, as discussed later on.

Module Setup

sim_type="spice"	determines which simulator will be invoked during simulation: spice, verilog, act
file="my_filename"	name of file with circuit description
instance="top_level_name"	name of top-level instance
file_type="act"	description language used: spice, verilog, act
name="my_name"	arbitrary name for COSIM's internal purpose and measurement reports
uses_blackbox="0"	use this option only if foundry cells with no transistor-level descriptions are used in circuit module

Few examples of the simulation types follow below.

sim_type ="spice" and file_type="act" : run netgen and VCS/HSIM cosimulation

sim_type ="prsim" and file_type="act" : run PRSIM/VCS cosimulation

sim_type="verilog" and file_type="verilog" : run only VCS simulation

Instance name is used in Netgen to specify the top-level instance and is a required parameter. Channel type, does NOT need to be specified, since COSIM parses all the ACT *defproc* definitions, as mentioned previously.

Once this information is provided for top-level module(-s), designer writes directives for all the necessary module and environment connections; specifies the mandatory parameters, such as paths, tool configuration files locations; specifies the optional parameters if desired; and runs the simulation by invoking COSIM and passing cosim.xml as the only parameter. COSIM performs the rest of initialization and setup automatically.

Once the designer invokes COSIM, the cosimulation initialization begins. If there were any problems found in cosim.xml, or any other tools (like Netgen, TLINT, etc.), COSIM immediately outputs this error to the user. If the error leads to termination of any of the tools, COSIM will stop the setup and inform the designer of a terminal error. If the error was not terminal, COSIM will print it to the screen and into a log file, and keep going either until a terminal error occurs or until cosimulation finishes.

As the cosimulation proceeds, COSIM will report the time progress of the simulation, as well as any events that happen at the environment's sources and sinks. Most of the information outputted by COSIM and the simulators goes to the screen and to the log files. A brief description of all the important output files will be provided in a later section.

Once the simulation completes, COSIM will run any post-processing trace file manipulations, if the designer selected this option. For example if designer selected to run TLINT, a *.trace and *.names files will be produced and TLINT will be initialized once the cosimulation completes.

If the user selected to perform frequency and/or power analysis in cosim.xml, COSIM will produce a file called hsim.mt that will contain all the results from the power and throughput computations performed in HSIM. Presently, the user may select a time window of when to perform such analysis (reset time is usually omitted from such computations). The user may select to calculate node capacitance (set to all nodes calculation by default), subcircuit power calculation (static and dynamic, depending on the input signal pattern), node voltage monitoring. By default the voltage values of all nodes is calculated and recorded by HSIM into the *.fsdb file format. This allows the designer to view waveforms of all nodes after the simulation finishes using CSCOPE waveform viewer or any other compatible viewer.

If power/frequency analysis was chosen, COSIM will setup HSIM and TLINT to output the results of all computations. Total circuit average and maximum current and power, subcircuit average and maximum current and power, subcircuit node average and maximum voltage, node capacitances will be reported by HSIM. The results of TLINT run, including slew rate violations, incomplete transitions, charge sharing problems, throughput will be reported back to the user as well.

Besides the above measurements, the user may select to perform additional checks in cosim.xml. These checks are performed in HSIM and assume that at least part of the circuit is implemented at transistor-level. These checks include bulk forward bias check, diode junction check, large current check (value, period), rise/fall time check, high impedance node check, inactive node check.

In order to run Netgen and TLINT, the user must specify the paths to Netgen's and TLINT's configuration files in the appropriate fields of cosim.xml. Also to correctly output TLINT compatible trace file, the designer must have a proper library libALINT.so in the local directory, where COSIM is run. The environment.v file with all the environment library elements must reside in the same location as cosim.pl (COSIM's executable). If the designer wants to use a cosimulation with PRSIM, appropriate vpi.o file must reside in the local directory as well.

Next few sections will go over some mandatory and optional parameters as well as how to perform environment connections and how to create interactions between various modules in cosim.xml.

3.6 Common Setup Parameters

As described in the previous section, COSIM is controlled through a unified interface XML-based file. In this file we specify mandatory and optional parameters that control all simulators (PRSIM, VCS, HSIM), as well as setup all the post-

simulation measurements and analysis.

Mandatory parameters include settings that are essential to the VCS/HSIM/PRSIM cosimulations. These simulators will not function properly if these parameters are not specified correctly. Among these parameters are:

- location of cosim.pl, which is the PERL-based executable for COSIM;
- global Vdd value (this value **must** match the value specified in the TLINT configuration file, otherwise the simulation will fail);
- *simulation time*, which is used both in VCS and in HSIM setup;
- *Verilog time scale*, which specifies the default time unit for all the values past to Verilog and the time precision with which HSIM and VCS are run (smaller time step leads to slower, but more accurate simulation);
- location of Netgen and TLINT configuration files (as described previously);
- length of _pReset and _sReset, which control the asynchronous reset, _sReset pulse should be longer than the _pReset pulse (and should enclose _pReset) to make sure that all asynchronous circuits are in a correct state, before they go into operating mode.

Additional parameters worth mentioning separately are the settings that have to do with the correct HSIM cosimulation setup:

- measurement start time signifies to HSIM when to start recording values for power/frequency measurements (typically after circuit reaches steady state, if applicable);
- measurement stop time signifies to HSIM when to stop recording values for power/frequency measurements (can be the end of simulation);

- device models are either passed to HSIM as a .spi file or a .lib file, depending on what kind of models are supplied by the foundry;
- device model type specifies which models should be used, i.e. TT; if this option is provided in the model file designer can select typical, fast, slow, etc.;
- *model aliases* set name equivalence classes between model names used in the library files and model names used in the spice files that were generated by Netgen or by another tool that was used
- HSIM globals, other than Vdd and GND, should be listed if more than one supply is used in the circuit, e.g. VSS;
- HSIM trace output type specifies the HSIM trace file format: *fsdb* is used by waveform viewers, such as CSCOPE, *alint* is used by TLINT post-processor;
- module inactive current specifies the current, below which HSIM may start ignoring the subcircuit in terms of transistor modeling, and assume that the subcircuit is inactive.

A full list of all important parameters, along with their brief descriptions, is presented in the tables below. For more detailed information regarding the HSIM internal variables that are not described below, please refer to the Synopsys user manuals that can be found on Synopsys Solvnet website.

Mandatory Parameters

PARAMETER	DEFAULT	DESCRIPTION
cosim.pl_path		path to automatic environment generator cosim.pl
vdd		global Vdd value for the circuit
sim_time		length of your simulation
timescale		VCS timescale
netgen_cfg		netgen configuration file name with path
_sReset_time		sReset length in time units (ns)
_pReset_time		pReset length in time units (ns)
measurement_start_time		simulation time when designer wants to start taking measurements
measurement_stop_time		simulation time when designer wants to stop taking measurements for power, etc.
insert_inverters_on_outputs	no	*yes* if you wan to perform power analysis using 2 inverters on interface channels with separate power source
inverter_size_um		takes effect only if *yes* to previous parameter; pmosWidth: pmosLength: nmosWidth: nmosLength in microns for each of the two inverters inserted on every Verilog output

Optional Simulation Setup Parameters

device_model_lib		device model library if using a spice .lib format
device_model_type		device model type if several are provided, for example TT
additional_spice_files		use this if you need to include addition spice files; can also be used for models if the format is appropriate
nmos_model_alias		specify aliases for nmos transistors, use spaces; *order matters*: usually netlist-file names first, model-file name last
pmos_model_alias		specify aliases for pmos transistors, use spaces; *order matters*: usually netlist-file names first, model-file name last
hsim_globals	Vdd, GND	any global signals, like vcc, vsin; omit Vdd and GND, as these two are declared by default

Optional HSIM run Parameters

HSIMOUTPUT	fsdb&alint	trace file type, fsdb used for Cscope, alint used for TLINT; fsdb&alint-both are produced with this setting
HSIMSPICE	3	device model accuracy: 0..3 (3-most accurate, 0-least accurate); to speed up simulation, the MOSFET model can be simplified
HSIMSPEED	1	simulator speed & precision; the value can be any integer from 0 to 5; higher speed values cause faster simulation speed at reduced simulation precision
HSIMANALOG	1	-1..3, controls the complexity of analog simulation algorithm; the higher the value is, the more precise and time-consuming the analog simulation algorithm will be
HSIMALLOWEDDV	0.1	the time step size is dynamically adjusted so that each node voltage change over the time step is limited by the value defined by HSIMALLOWEDDV
HSIMITERMODE	1	the HSIM iteration control parameter; can be set integer from 0 to 2; 2-highest accuracy, 0-lowest

Optional HSIM run Parameters Continued

HSIMSTEADYCURRENT	10n(A)	idle circuit (skipped by simulator) current; a subcircuit is treated as being idle if every node in the subcircuit is idle
HSIMIGISUB	0	0..3, 1=all_on, the static gate leakage currents and the substrate current in the calculation of I-V result for a MOSFET transistor with the BSIM4 model; *note:* if HSIMSPICE is set to be greater than 0, HSIMIGISUB=1 by default
HSIMCHECKMOSBULK	1	0(off)..1(on), check to identify potential forward bias conditions in MOSFET parasitic diodes. HSIM will report Warnings when the bulk of a NMOSFET transistor can potentially be greater than 0.5V or the bulk of a PMOSFET transistor can potentially be smaller than 0.5V
HSIMDIODECURRENT	1	0..2, when HSIMDIODECURRENT is set to 1 or 2, the dc current of MOSFET parasitic diodes will be calculated

Additional Checking Options

large_current_check	10m(A)	current check: warn if supply current exceeds this value
large_current_period	1n(s)	period for large current checks (frequent checks drastically slow down the simulation)
rise_time_check	.01n(s)	value for excessive rise time check
fall_time_check	.01n(s)	value for excessive fall time check
excessive_rise_fall_fanout	1	0..2; 1=driving trans; defines the fanout of driver nodes; if fanout is set to 1, only the driver nodes with fanouts are checked to avoid unnecessary checks on internal nodes within logic gates; if fanout is set to 0, both the internal nodes and the driver nodes are checked; the default value is 0 (fanout=0: all nodes; fanout=1: nodes that have direct connection to transistor's gate; fanout=2: nodes that have direct connection to transistor's bulk)
high_impedance_fanout	0	0..2(same as above); fanout=0: all nodes; fanout=1: nodes that have direct connection to transistor's gate; fanout=2: nodes that have direct connection to transistor's bulk

Measurement Options

subcircuit_names_for_pwr_meas	Vdd	specify nodes for average and maximum power measurements; results reported in hsim.mt, use spaces; ACT syntax (no x-s in front of subcircuit names)
subcircuit_node_voltage		specify nodes for average and maximum voltage measurements; results reported in hsim.mt, use spaces; ACT syntax (no x-s in front of subcircuit names)
HSIMNODECAP	*	specify nodes for capacitance calculations; *=(all nodes). the average capacitance of the specified node_pattern is reported and stored in the capacitance report file hsim.cap
tlint_config_path		specifies path to TLINT configuration file for post-simulation analysis
run_tlint	no	if set to *yes* runs TLINT analysis at end of simulation; *no* - disables TLINT

Other Options and Commands

HSIMNTLFMT		specifies the netlist format of the input file. It is an alternation to -netlist_format command line option. It is a global parameter and its default value is hspice format
HSIMTOP		specifies the top-level subckt name (if required). It is an alternation to -top command line option. It is a global parameter and its default is null
hsim_param		if you want to set additional HSIM parameters, type exact syntax of the spice command as you would in a spice setup (or netlist) file
HSIMRISE	0.001V/ns	default value for rise time of signals coming from Verilog
HSIMFALL	0.001V/ns	default value for fall time of signals coming from Verilog

3.7 Sources and Sinks Library for Testing Environment

All elements of the environment in COSIM are created as Verilog-type modules. These building blocks may perform various tasks including completing asynchronous 4-phase handshakes, logging output values and comparing them to a predefined set of values. Sources and sinks also perform some logical correctness checks (such as mutual exclusion) and timing checks (such as sequence checking of data and enable arrival times).

The list of all sinks and source available in COSIM is presented in the tables

below. The tables describe the types of sinks and sources; what kind of channels a given source/sink is able to connect to; their corresponding XML names as used in cosim.xml; connection syntax for these environmental elements. For sources, we also describe what values will be sourced; for sinks, we describe how to perform value checking on the output channels.

Available Sources

Type	Channels	Name in XML	Values Sourced	Connection
random	e1ofN, ev1ofN, eMx1ofN, evMx1ofN, bool	type= "random"	seed ="132"	channel="line1", *for bool add:* clk_used="clock"
inject	e1ofN, ev1ofN, eMx1ofN, evMx1ofN, bool	type= "inject"	inject_file ="y.dat"	channel="line1", *for bool add:* clk_used="clock"
constant	bool	type= "constant"	value="_sReset"	channel="reset"
clock	bool	type= "clock"	rate="650" (in MHz)	channel="clock", reset_signal= "reset"
globals	all globals	type= "globals"	all globals	channel="g"
piece-wise linear	bool	type= "pwl"	filename ="y.dat"	channel="in"

Note:

- Inject files must have the number of values on the first line, then one value per line;

- For the piece-wise linear source, y.dat is a file that contains: Line 1 : n = The number of values; Line 2 : Initial value of the signal; Line 3 to Line 3+n: Delays to toggle the signal;
- Constant values are supported by *inject* source by setting (constant =1), by *constant* source which can assign single value or value of another wire by setting (value=1), and by *piece-wise linear* source, if properly specified in y.dat;
- *clock* source uses a reset signal as a starting point for the clock pulses.

Most commonly used sources in our designs are *random_source* with uniform, exponential, and normal input probability distributions; and *inject_source* that uses values specified in an injectfile. The values can be specified in base-10 format; there is no need for binary representation, since the source automatically performs this conversion to properly insert the given values on a specified channel. All channel types are supported by these two sources.

Available Sinks

Type	Channels	Name in XML	Comparison Values	Connection
just sink	bool, e1ofN, ev1ofN, eMx1ofN, evMx1ofN	type= "sink_only"	none	channel="outp", *for bool add:* clk_used="clock"
watch and compare	bool, e1ofN, evMx1ofN	type= "watch_N_compare"	expected= "exp.dat"	channel="outp", *for bool add:* clk_used="clock"
watch and sink	e1ofN, ev1ofN	type= "watch_N_sink"	expected= "exp.dat"	channel="outp"

Note:

- *just sink* only logs outputted values and controls enable for the 4-phase handshake in the asynchronous case;
- *watch and compare* compares outputted values to the expected values in a given file and does not toggle enable;
- *watch and sink* compares outputted values to the expected values in a given file and controls enable for the 4-phase handshake in the asynchronous case;
- Presently all sinks are setup in such a way that COSIM automatically produces output_name.dat files with all the values that were logged from each given sink.

All of the listed sinks have been extensively used in many design created by our group members. *sink_only* is commonly used when designers are not interested in

the values produced and only care about the correct communication of the environment with the given channel in the circuit. *watch_N_compare* does not toggle the enable, which is commonly used in synchronous circuits, and in asynchronous circuits where the bucket is a part of the circuit and designer only needs to monitor the channel (and/or compare against given values) and not actively perform the handshake. *watch_N_sink* performs an active handshake in case of an asynchronous circuit and also has the ability to check the obtained values. It is trivial to extend the listed sinks to support other channel types, but so far we have not seen the need for it in the designs that we have been testing.

For source and sinks that support validity-based channels, i.e. evMx1ofN, the validity calculation is enabled by setting *(calculate_v = 1)* in cosim.xml. By default, this variable is set to 0 *(calculate_v = 0)*.

All top-module ports *must* be connected in cosim.xml, or explicitly left dangling. This is done to make sure that the designer does not forget to connect any of the important ports of the design. Main directives for environment connection in COSIM are source connection, sink connection, global variable connection and leaving a dangling port.

All the ports that were left dangling in COSIM, will also be reported by HSIM as unconnected nodes in a file called hsim.conn. The designer should always check that file upon successful completion of simulation (if this file is nonexistent, than there are no unconnected nodes), to make sure that only nodes that are indeed not supposed to be driven are listed in that file.

All sinks and sources are connected within the *circuit* module block and the syntax looks as follows:

$< source\ type = "random"\ channel = "l" / >$

$< sink\ type = "sink\_only"\ channel = "r" / >$

$< disconnect\ channel = "not\_used" / >$

The first line creates an instance of a *random* source on channel l; the second line creates an instance of a *sink_only* bucket with no value checking on channel r; and the third line keeps channel *not_used* explicitly disconnected. Outside angled brackets refer to XML's standard syntax.

If a given design slice has a very large number of inputs and/or outputs, it becomes cumbersome and time-consuming for the designer to create instance of sources and/or sinks for all of the ports in cosim.xml. To minimize the designer's effort in such cases, we have added an *auto-connect* option to COSIM (passed as a parameter in shell command *cosim -A*) to perform such connections automatically. If auto-connect option is selected, COSIM will automatically connect default sinks and source to the circuit, based on the channel types obtained from our JAVA-base parser. This greatly improves the design time / testing time for large designs with a high number of input/output nodes. Presently, in the auto-connect mode, COSIM by default connects *random* sources based on uniform probability distribution and *just_sink*-type buckets with no value comparison. These elements are the most universal out of the implemented ones and they can connect to any synchronous and asynchronous channel type.

3.8 Global Connections

Besides directives for module setup, COSIM's general setup, circuit environment connection and simulator parameters, COSIM also includes calls for making intra-module and global variable connections. With the help of these directives, designers can connect various modules of the design. These modules may be described in different hardware languages (Verilog, spice, ACT) at different levels of abstractions (behavioral, stack-based, standard-cell based, or transistor level). Besides

making connections between modules, designer can also connect (or explicitly disconnect) top-level ports to global variables (such as Vdd, GND, _sReset, _pReset, etc.).

In order to achieve such connections, we have introduced $<$ *connect* $>$ directives. There are several types of *connect types*:

- _sReset: Connects the signal to _sReset;
- _pReset: Connects the signal to _pReset;
- boolean: Connects a wire between signal specified by "from_channel" to signal specified by "to_channel".

These connections are performed outside of the module description XML block. An example of such connection would be the following line:
$<$ *connect type* $=$ "*_sReset*" *from_circuit* $=$ "*n*1" *from_channel* $=$ "*_Reset*"/ $>$,
where the outside angled brackets refer to XML's standard syntax.

This line creates a connection between _Reset port of circuit earlier labeled as n1 to a global variable, called _sReset. All connection between two different modules are created in a similar manner.

The $<$ *connect* $>$ command completes our set of introduced directives and allows the designer, along with the other directives, to create any arbitrary design description with synchronous and asynchronous circuit parts described in different hardware languages at various levels of abstractions and corresponding testing environments. Using the introduced syntax, COSIM reads in the circuit description and various tool parameters from cosim.xml; and automatically sets up all the simulators, pre-processing and post-processing tools for circuit's compilation from various description languages, design's simulation and analysis.

3.9 Output Files

If designer's HSIM simulation completed successfully, one should see the following files in the local directory where the designer ran the simulation. Information provided in these files has crucial effect on the circuit's correctness, performance, and manufacturability.

Important Output files

File Name	Description
environment_used.v	Verilog sources and sinks that are used in this particular design instance
hsim.ach	List of active nodes in the design
hsim.cap	List of all nodes and their calculated capacitances
hsim.chk	Power Check Results: Excessive Current Check, High Impedance Node Check, Excessive Rise/Fall Time Check
hsim.conn	**Dangling Node Report** (if this file is missing, all nodes are connected)
hsim.csintf	Digital-to-Analog and Analog-to-Digital Conversion channel list and event count
hsim.dcpath_err.chk	DC Path Check results on all nodes
hsim.fsdb	CSCOPE compatible trace file
hsim.hsimba	List of active nodes for back annotation
hsim.ina	Surrounding inactive node list without DC signals
hsim.ina_all	Complete inactive node list
hsim.log	Simulation Progress Log
hsim.mt	Text-format file with all the power, current, voltage measurements
hsim.trace	TLINT compatible trace file
inverter_separate_vdd.spi	If designer has selected to insert buffers on interface channels, this file will have the sized subcircuit for inverters used in the buffers

Chapter 4

Toolflow Evaluation

4.1 Benchmark Considerations

As stated earlier we have developed two sets of libraries for evaluation of our toolflow: a static synchronous library and QDI PCEHB-based asynchronous library. We have taken ITC-99 benchmarks and compiled them into synchronous and asynchronous netlists using our novel toolflow. ITC-99 benchmarks have been created and maintained with collaboration of several research groups [3], [2]. For transistor models we have used the educational Nangate 45 nm design kit [1]. The throughput measurements, as well as power analysis are presented in this chapter.

We have selected ITC-99 benchmarks, since most of them represent small to medium size realistic designs. The compiled (using Synopsys Design Compiler) benchmarks incorporate the following circuit structures: internal buses, input / output ports, global reset signal, single-frequency clock signal, D-type flip-flops, transistor-realizable circuit modules. All of these benchmarks are written using behavioral VHDL-level descriptions. The list of the tested benchmarks and their original functionality is shown in Figure 4.1.

Name	Original Functionality
b01	FSM that compares serial flows
b02	FSM that recognizes BCD numbers
b03	Resource arbiter
b04	Compute min and max
b05	Elaborate the contents of a memory
b06	Interrupt handler
b07	Count points on a straight line
b08	Find inclusions in sequences of numbers
b09	Serial to serial converter/encoder
b10	Voting System

Figure 4.1: ITC-99 Benchmarks and their Functionalities

All of the implemented benchmarks are circuits with variable complexity, developed for different types of applications. The internal structure (in terms of logical elements) of each benchmark is presented in Figure 4.2.

Benchmark	Gates	Flip-Flops	Inputs	Outputs	VHDL lines
b01	45	5	4	2	110
b02	25	4	3	1	70
b03	150	30	6	4	141
b04	480	66	13	8	80
b05	608	34	3	36	319
b06	66	9	4	6	128
b07	382	51	3	8	92
b08	168	21	11	4	89
b09	131	28	3	1	103
b10	172	17	13	16	167

Figure 4.2: ITC-99 Benchmarks Structure

One can observe that the largest designs contain around 600 gates with tens

of flip-flops. Some of these benchmarks are computation based, others - control based. It is for this reason that we have selected ITC-99 benchmarks. They offer a wide range of design complexities and design types for testing our toolflow.

4.2 Throughput Comparison

For the speed measurement of the asynchronously implemented benchmarks, we run the circuit with random input patterns for a sufficient amount of time to reach steady state and then measure it's average throughput. As for the synchronous circuit, we initially run it at a low frequency for a long period of time and record all the output values, using random inputs. We then gradually increase the clock rate of the circuit and compare the obtained values with the originally recorded values. As soon as there is a mismatch in the outputted values, we know that there was a timing violation somewhere in the circuit, i.e. setup or hold time of a flip-flop was violated. At that point the last correctly recorded frequency is the maximum circuit frequency under the given conditions.

Since none of the synchronously implemented benchmarks have explicit clock trees; we model the clock tree load, skew and jitter by giving the maximum operating frequency a 20 percent safety margin. The safety margin given to synchronous circuits is usually 20 percent or higher as reported in the literature [36]. From unofficial sources, we know that AMD uses a clock safety margin of 25 percent and Intel of 33 percent.

Below are two figures that show the throughput comparison of the synchronous and asynchronous implementations. Figure 4.3 shows absolute throughput measurements in MHz with random input patterns. Figure 4.4 plots the asynchronous circuit frequency, normalized to the synchronous circuit frequency, where 1.0 represents the synchronous frequency per benchmark.

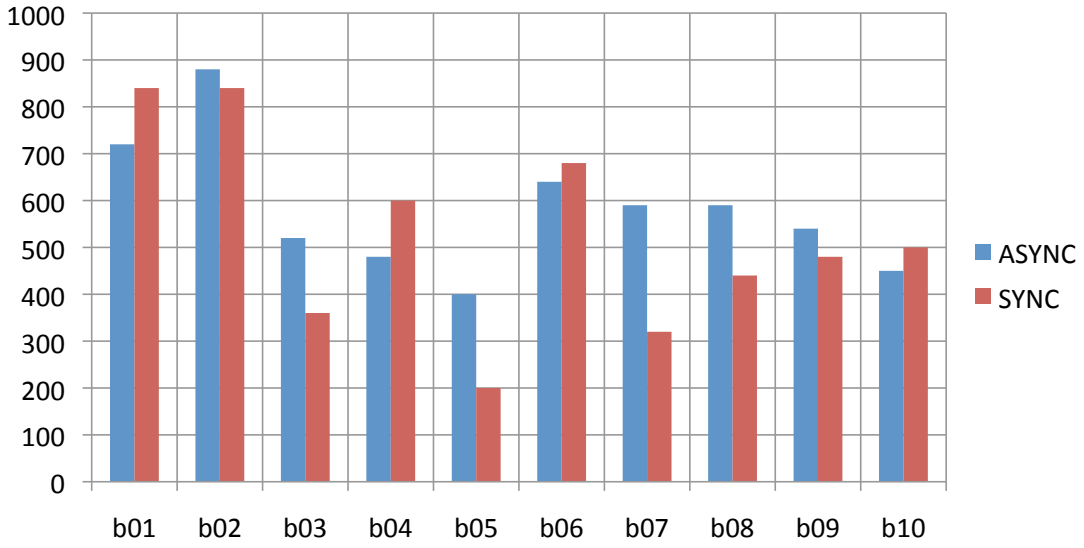

Figure 4.3: ITC-99 Benchmarks: Synchronous and Asynchronous Implementations in MHz

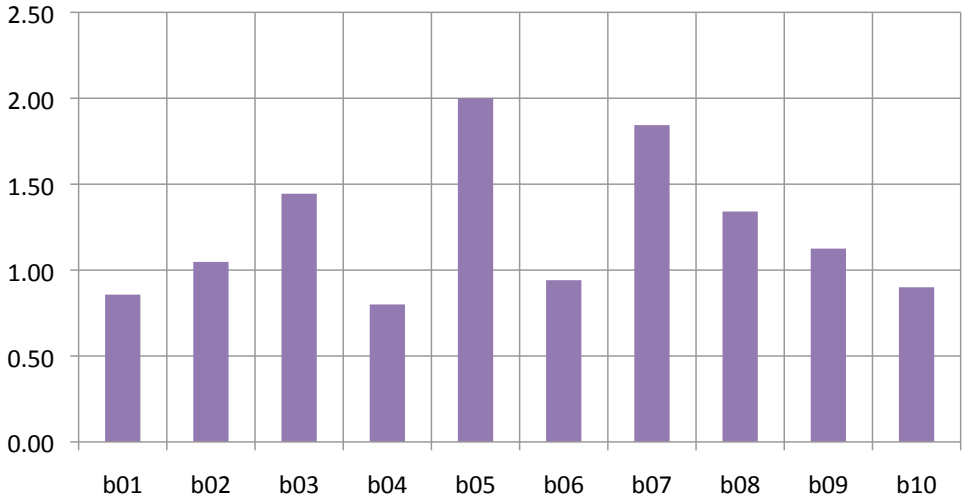

Figure 4.4: ITC-99 Benchmarks: Asynchronous Throughput Normalized

In both sets of simulations, we have implemented ideal environmental behavior (inputs, outputs, clock). All the transitions from the environment are monotonic with infinite drive strength and very fast slew rates. All internally unconnected nodes were bucket-ed in the asynchronous implementations. The synchronous reset network was replaced with asynchronous _sReset and _pReset networks.

4.3 Process Variations

In order to analyze the impact of foundry process variations incurred due to typical device mismatch, we run the same set of synchronous and asynchronous simulations in two process corners: slow PFET - slow NFET (SS), and fast PFET - fast NFET (FF). Figures 4.5 and 4.6 demonstrate the effect that process corners have on the behavior of synchronous and asynchronous circuits respectively.

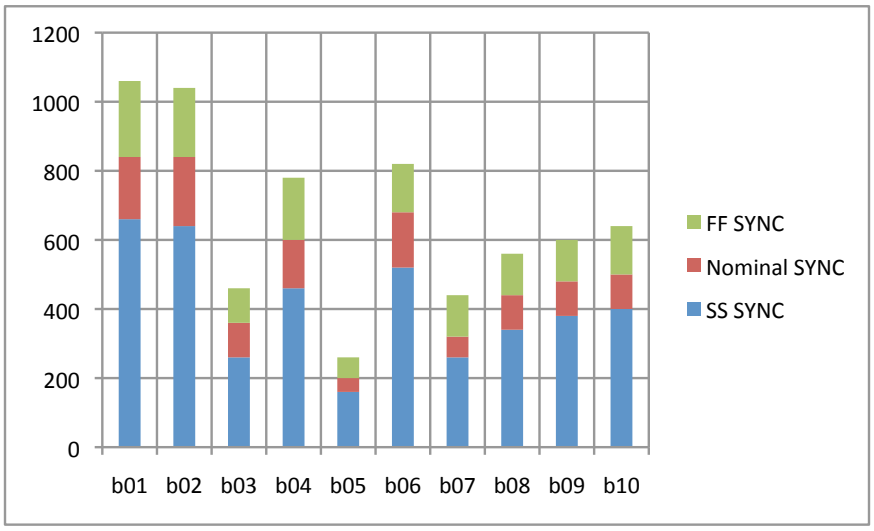

Figure 4.5: Process Variations: Synchronous Implementations

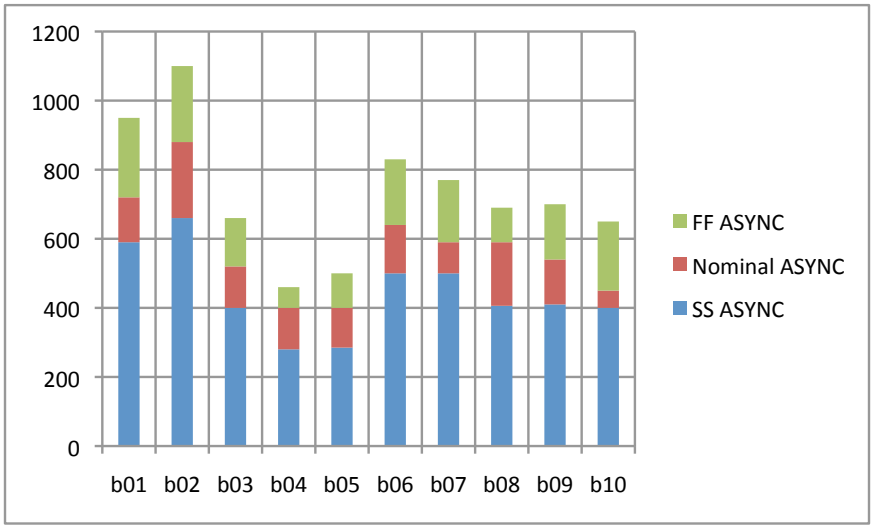

Figure 4.6: Process Variations: Asynchronous Implementations

As expected, the throughputs of both circuit families increase as the NFET and PFET devices get faster. For comparison purposes, we demonstrate the throughputs of both families in all process corners in Figure 4.7. All the simulation parameters, other than the device models, are kept consistent with the nominal device model simulation setup. The synchronous benchmarks are still given a 20 percent clock margin from the first point of failure, as previously explained.

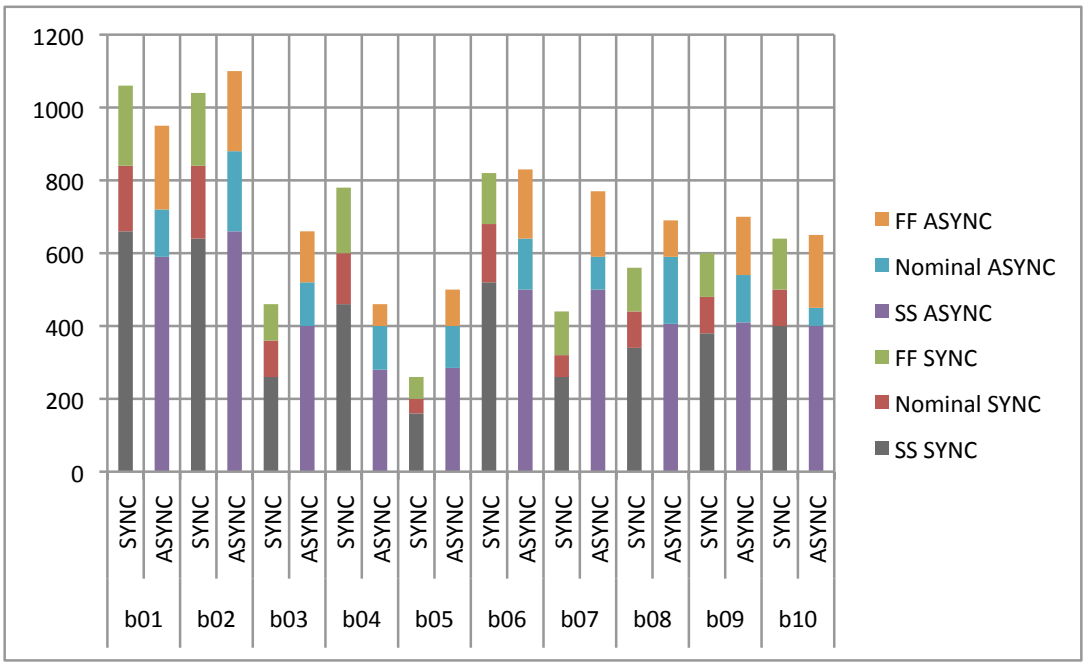

Figure 4.7: Process Variations: SYNC and ASYNC Implementations

The trends for the SS and FF corners exactly resemble the nominal scenario in Figure 4.3. Such behavior confirms that all the analysis, as well as all the conclusions drawn from our experiments with the nominal device models can also be used in the presence of process variations. The process variations considered in this study are the 3-sigma variation, represented by the two process corners.

4.4 Power Analysis

In all the conclusions related to power analysis, due to the implementation of the benchmarks, we have neglected the clock tree load and switching energy. If the clock network was taken into account, the power consumption of the synchronous circuit would drastically increase.

As expected, due to the nature of our transformations (gate level and not process level), the power consumption of the asynchronous implementations is much higher than that of the synchronous (with the clock tree neglected).

However, in the applications where input activity is variable, specifically, high-frequency bursts of inputs followed by long periods of inactivity in the inputs, asynchronous circuits provide power savings. Such applications include speech processing, on-chip networks, neurobiological circuits, etc. In such designs, the circuit implementations must still be able to support maximum throughput, thus, the clock has to run at maximum rate in the synchronous implementations.

In our next set of experiments both synchronous and asynchronous frequencies were fixed (per benchmark) to the highest frequency that both implementations were able to support. We then effectively vary the input signal activity factor by sending high frequency bursts of inputs and long periods of input inactivity following them.

At the scale of benchmarks in our analysis (small and medium size circuits), clock-gating overhead is extremely high due to the small size of the circuits. We, thus, compare the asynchronous implementations to non clock-gated synchronous implementations with the clock distribution network ignored.

Figure 4.4 demonstrates the break-even average frequency of when the power consumption of the asynchronous implementation is equal to the power consumption of the synchronous implementation. As mentioned previously, the inputs are

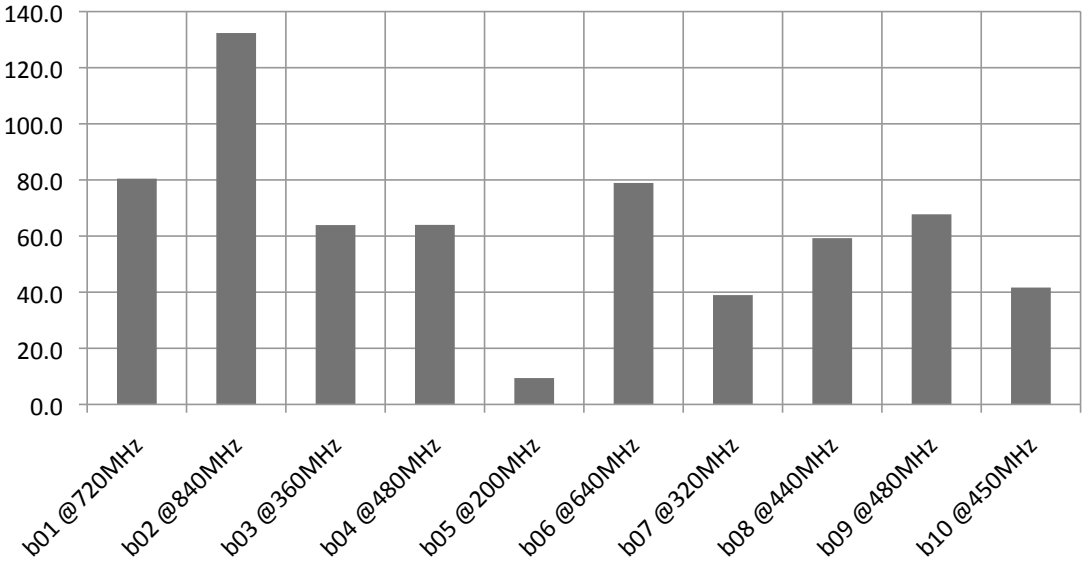

Figure 4.8: ITC-99 Benchmarks: PWR break-even frequency in kHz

supplied in bursts, so this average frequency (in kHz) approximately designates how often these bursts occur for the dynamic power consumption of the two implementations to be the same. If clock distribution network is taken into account, these average frequencies will naturally become higher, providing a better trade-off. The computation is performed in the following manner:

$$P_{async} = C_{total} Vdd^2 f_{sw} = \alpha C_{total} Vdd^2 f_{avg} = C_{total} Vdd^2 f_{avg}$$
$$P_{sync} = C_{total} Vdd^2 f_{sw} = C_{cl} Vdd^2 f_{sw} + C_{clk} Vdd^2 f_{clk} = \alpha_{cl} C_{cl} Vdd^2 f_{clk} + \alpha_{clk} C_{clk} Vdd^2 f_{clk}$$
$$P_{sync} = 0.5 C_{cl} Vdd^2 f_{clk} + C_{clk} Vdd^2 f_{clk}$$

$P_{sync} = P_{async}$ at a set average input frequency (f_{avg}) and fixed clock rate (f_{clk}):

$$0.5 C_{cl} Vdd^2 f_{avg} + C_{clk} Vdd^2 f_{clk} = C_{async} Vdd^2 f_{avg}$$
$$0.5 C_{cl} f_{avg} + C_{clk} f_{clk} = C_{async} f_{avg}$$
$$0.5 C_{cl} f_{avg} - C_{async} f_{avg} = -C_{clk} f_{clk}$$
$$f_{avg} = \frac{C_{clk} f_{clk}}{C_{async} - 0.5 C_{cl}}$$

This power calculation assumes comparable leakage power consumption for the synchronous and asynchronous implementations. In that case, power consumption is dominated by dynamic power. Such assumption proves to be correct in low leakage technology, where both drain-to-source and gate leakages are minimized.

In the provided derivation, the activity factor, α, in asynchronous power calculation; similar to α_{clk}, is equal to one, since all triggered output nodes of the asynchronous circuit switch twice per token (once to active phase and once to neutral phase). The clock nodes follow the same trend. α_{cl} represents the activity factor of synchronous combinational logic and is estimated around 0.5 for computational purposes. Using these facts, we calculate synchronous and asynchronous power consumptions and equate them for a fixed clock rate, which indicates the maximum frequency of inputs within the high-frequency bursts (i.e. the maximum throughput that the given benchmark supports).

The obtained result shows that the average break-even frequency f_{avg} is directly proportional to the maximum clock rate f_{clk} and clock network capacitance C_{clk}; and inversely proportional to the difference of total asynchronous capacitance C_{async} and fraction of clock tree capacitance C_{clk}, as expected. In derivation of Figure 4.8, all the capacitances were extracted from the actual circuit, however, the additional capacitance coming from the clock tree distribution network was omitted from the calculation.

4.5 Design Space Analysis

To give the reader a slightly better understanding of the synchronous / asynchronous tradeoffs presented in this thesis, we present a study that portrays a full deign space view for one of the analyzed benchmarks. Figures 4.9 and 4.10 present a 3-dimensional view of the maximum supported throughput vs. average input

burst frequency vs. calculated dynamic power consumption of the asynchronous and synchronous implementations respectively.

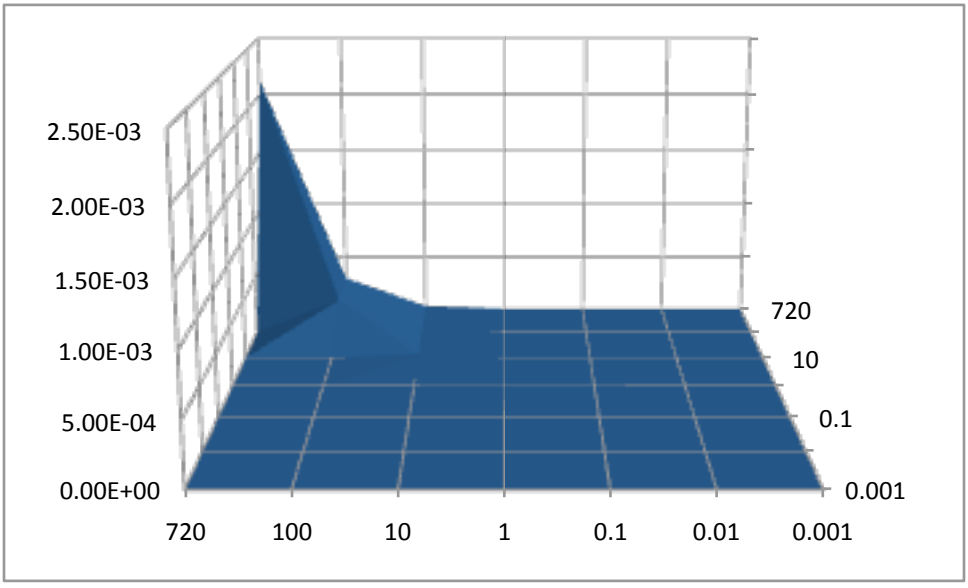

Figure 4.9: Power for b01 in Watts: Asynchronous Implementation

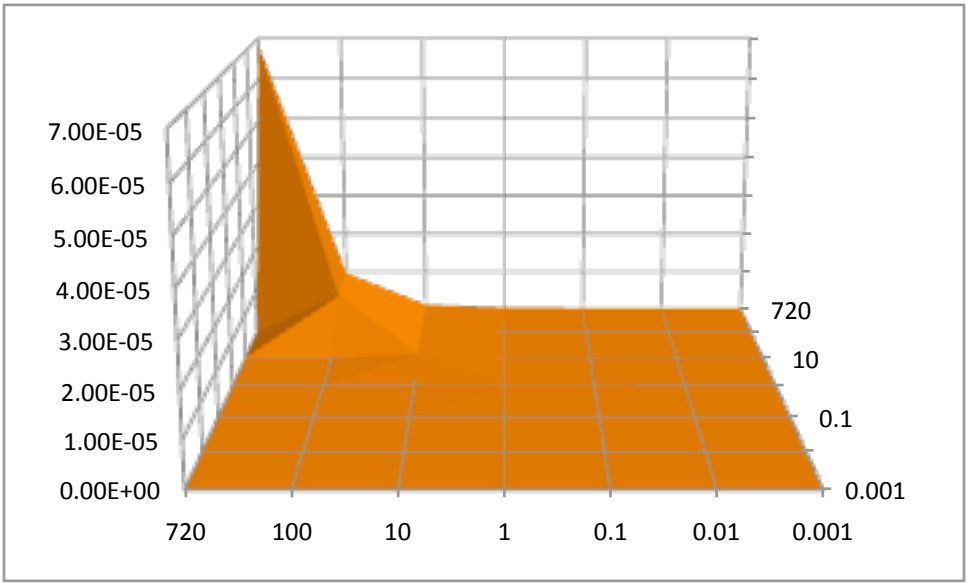

Figure 4.10: Power for b01 in Watts: Synchronous Implementation

In all charts, x-axis (horizontal) represents the maximum supported frequency for b01. The third dimension, z-axis indicates the average frequency of input

signals bursts, as described in the previous sections. Y-axis (vertical) represents the power consumption, given the maximum frequency and the average input frequency (input signal duty cycle).

In these surface plots, the areas where the average input signal frequencies are higher than the maximum supported frequency, should be ignored. These are merely artifacts of the software used for the plotting purposes.

Next, in order to have a better comparison of the two implementations, we subtract the power consumption of the synchronous implementation from the power consumption of asynchronous implementations for all frequency ranges. The resulting plot is demonstrated in Figure 4.11.

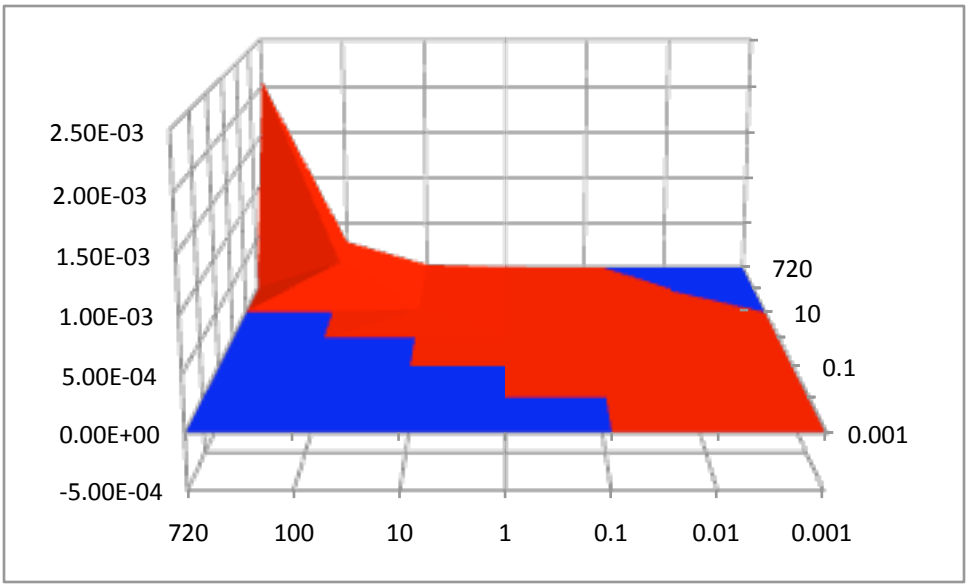

Figure 4.11: Power for b01: Diff between ASYNC and SYNC Implementations

As previously stated, the unattainable throughput scenarios should be ignored. For the rest of the states, the synchronous implementation provides a more efficient power consumption at higher input signal average frequency (indicated by the red surface); and asynchronous implementation provides a better power consumption tradeoff for the areas where high maximum throughput needs to be supported, but

the average input signal frequency is much lower (indicated by the blue surface), as discussed in Section 4.4.

This result agrees with the original hypothesis that we have made, while developing this synchronous-to-asynchronous transformation. Our prognosis stated that due to the nature of the transformation (gate-level netlist conversion), the number of transistors in the asynchronous implementation would on average be higher than that of the corresponding synchronous implementation. Also, in the asynchronous QDI 4-phase handshake circuits, the signal activity factor is higher, since the circuit on every data token goes through the active phase and then returns back to the neutral phase. These two facts lead to higher dynamic power consumption of the ASYNC design when the inputs arrive at high frequency with no inactive periods. However, due to data-driven nature of the QDI implementation, in the scenarios where the average input frequency is low, but bursty with a high maximum input frequency within the burst, asynchronous circuits prove to be beneficial in terms of dynamic power consumption. This mainly occurs due to the ASYNC automatic shutdown property in the time periods when the inputs are inactive.

The graphs for the rest of the benchmarks look almost identical and will not be included here, since they introduce no novel information compared to the presented plots. The only difference in the plots of the other benchmarks are the eaxct values on the break-even frequency line, as shown in Figure 4.8.

4.6 Designer Guidelines

After carefully analyzing the obtained results, we have developed the following guidelines for the designers to consider, when their designs are still at either behavioral level of abstraction or at the gate level netlist. The following two lists

provide some rules of thumb of when synchronous and asynchronous circuits should be used, depending on the metrics that are targeted.

SYNCHRONOUS CIRCUITS

- Ideal for very small pipelines stages
- Simple functions with small number of inputs
- Constant rate, high-frequency inputs
- Circuits with many high-fanout nodes (wire copy is free in terms of utilized transistors)
- Constant frequency of system operation

ASYNCHRONOUS CIRCUITS

- Ideal for medium size circuits
- More than 16 transitions between original pipeline stages (typically 40+ gates on critical path)
- Functions that need multiple stages in static CMOS for implementability
- Complex functions with both positive and negative senses of inputs/outputs
- Inputs with long periods of inactivity followed by high frequency bursts
- Variable frequency requirements

Chapter 5

Asynchronous-to-Synchronous Interface with Discrete Timing

5.1 Motivation and Background

Many modern systems contain both synchronous and asynchronous circuit parts in order to achieve the required design metrics. In such systems, *correctly* crossing synchronous/asynchronous boundary is a non-trivial problem. An example of a system with synchronous/asynchronous divide would be GALS (globally asynchronous, locally synchronous) systems that were first suggested by Chapiro in 1984 [8]. Such designs may contain two types of boundaries: synchronous-to-asynchronous and asynchronous-to-synchronous. Next few sections describe these two bonders in detail; we give an overview of the work that has been previously done by various researchers in this area.

5.1.1 Synchronous-to-Asynchronous Boundary

In order to go from synchronous domain to asynchronous domain, one has to guarantee that the minimum throughput of an interfacing asynchronous circuit is higher than the clock rate of the interfacing synchronous circuit. The asynchronous circuit has to be able to accept data tokens faster than the synchronous circuit produces these tokens (asynchronous circuit has to be working faster than the clock rate). Asynchronous part of the interface has to work without getting backed-up and without stalling.

Some interfaces are more demanding than others and require the asynchronous circuits at the interface to not only be faster than the corresponding synchronous circuits, but also to deal with the problem of incoming synchronous signals changing at an arbitrary time (without any relationship to the other synchronous or asynchronous inputs and without correlation to asynchronous system's internal state). Such interfaces assume no prior knowledge of the synchronous system's behavior; more than one inputs may arrive simultaneously, or immediately after each other (could be both synchronous and asynchronous in nature). The problem with this situation is that the interfacing asynchronous circuit may enter a metastable state at some output node (for example, if more than one inputs arrive simultaneously), which could cause the entire asynchronous part of the system to deadlock. This situation has been extensively studied; many solutions have been derived, with asynchronous synchronizer being the most attractive and simple approach [27].

As discussed by Nystrom, one the most obvious ways to handle multiple simultaneous requests (arrival of several inputs that are not temporally correlated) is to use a mutex element (also called arbiter) [13, 20, 32]. A simple two way arbiter can be constructed with a pair of two cross-couple NAND gates, as shown in Figure 5.1.

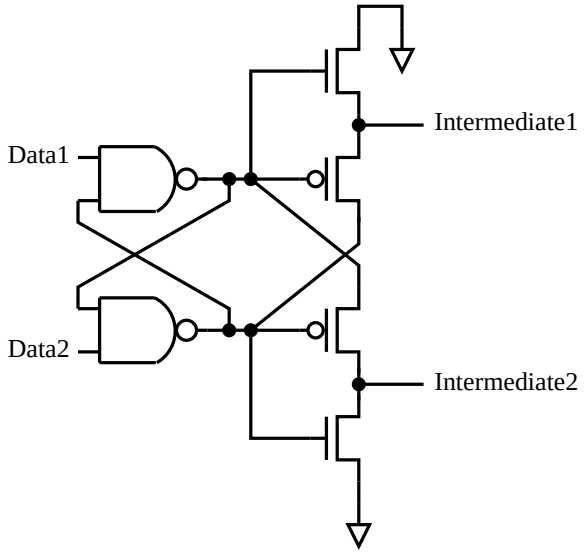

Figure 5.1: Two-Way Arbiter Structure with filter

In this configuration, no matter what the input combination is, only one of the outputs (either $Intermediate1$ or $Intermediate2$) is high at any given time (guarantees mutual exclusion of the input requests). The average arbitration time of the circuit is proven to be constant (if no initial conditions are set and both inputs: $Data1$ and $Data2$ become simultaneously high). The proof is obtained by using transistor physics to determine the output voltages [20].

The filter at the output of the cross-coupled NAND pair eliminates any non-monotonicity in signal transitions and all the values that cannot be interpreted as valid logic values. The filter is inverting and waits for the signals to be separated by at least a threshold voltage before the output changes.

In the case when multiple synchronous requests enter the synchronous / asynchronous boundary, a way of managing them would be to store these requests in modified flip-flops, which will be discussed later in this chapter. Such flip-flops will then produce proper asynchronous tokens at their outputs. In that scenario, we still have to deal with the issues of these signals not being mutually-exclusive as seen by the asynchronous system, however, they can now be treated in a fully-

asynchronous manner. In this case, the flip-flops are designed in such a way that these simultaneous request signals can clear their states (and flip-flop stored states) based on the value of the acknowledge signal coming from the asynchronous part of the system.

In this situation, we could use the following modified mutex element that can treat simultaneous arrival of two non-mutually exclusive asynchronous requests. The modified flip-flops would be located immediately to the left of this arbiter circuit.

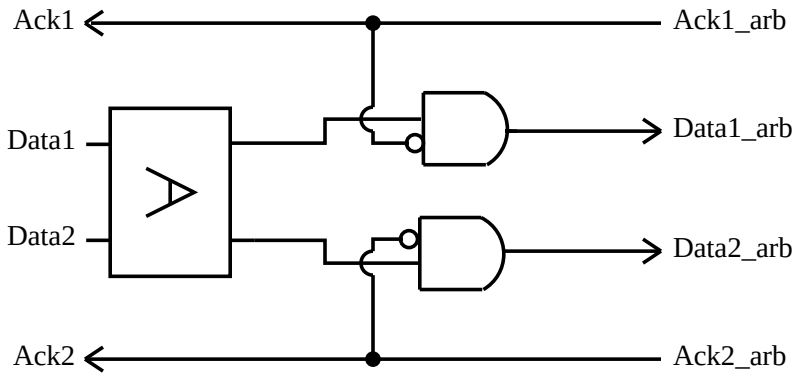

Figure 5.2: Mutual Exclusion Element with QDI handshake

In order to implement a mutual exclusion element that will not violate any of our four-phase QDI handshakes, not only the requests themselves, but also the acknowledges that indicate that the requests were served have to be utilized in the circuit. The basic mutual exclusion element for the applications in asynchronous circuits is shown in Figure 5.2.

These elements may be cascaded to obtain arbiters of higher dimensionality, i.e. arbiters that can handle more than two simultaneous requests. Techniques of cascading mutex-s have been previously analyzed by Manohar [19].

5.1.2 Asynchronous-to-Synchronous Boundary

Going from asynchronous part of the design to synchronous part can be more complicated. This is the area where we want to focus our attention. Many solutions have been previously proposed. Among them is another type of *synchronizer* (standard two flip-flop) solution, which is simple, elegant, but has a large latency (and in some cases throughput) penalty and does *not* satisfy some of the requirements of more complex interfaces. Also no useful logical operations may be performed while the interface signals are being synchronized. A lot of theoretical and experimental work has been done to test the flip-flop approach at the synchronous/asynchronous boundary. Probabilities of metastability propagation and circuit failure have been carefully analyzed [21, 38]; it has been shown that sequential connection of flip-flops greatly decreases the metastability propagation. As a matter of fact, only two sequential flip-flops drive this probability to an almost zero-value. Much experimental work has been performed to demonstrate such behavior in actual industrially-used circuits [7, 29, 31].

Some more complex schemes were presented to drive the failure probability down while using two flip-flops as the main synchronization circuit, with some additional circuits at the periphery. One approach is using stoppable clocks while performing the synchronizations between two domains [17, 26]. Such approaches require more hardware as well as calibrated delay lines, but can be used in conjunction with other techniques if desired. Various arbiter configurations and asynchronous clock-pulse generators have been previously used at the interface of the synchronous/asynchronous boundary. However, sometimes, the synchronous requirements don't allow clock stoppage or irregular clock periods. Also, all of the above approaches require overhead hardware, and in many cases higher-level control circuits.

Additional synchronization approaches have been proposed, such as pipeline synchronization technique [32]. Using this method, several pipeline stages are used to perform synchronization of the asynchronous *request* signal with a synchronous clock. This technique uses arbiters and tries to reduce the timing penalty compared to the consecutive flip-flop synchronization. The issue is that such implementation would most of the time require insertion of redundant pipeline stages (and arbiters), which increases area and power consumption.

Another commonly used approach is dual-port FIFO placement at the interface. This method is used not only at the synchronous/asynchronous boundaries, but also at synchronization of circuits that use several clock domains. This technique is safe in terms of metastability issues (as long as the FIFO depth is properly calculated), however, it has a clear area/power cost and has relatively high latency penalties in some modes of operation (when the FIFO is completely full, or completely empty). It is most efficient when the *read* and *write* rates are almost equal, which is not the case in most designs.

5.2 Additional Asynchronous-to-Synchronous Interface Requirements

Besides the requirements for the asynchronous-to-synchronous interface described above, we have discovered that a lot of times we would like to put more constraints on the interface. Specifically, in applications of distributed sensor networks and in neuromorphic systems, there is a necessity to not only synchronize the data traveling from asynchronous domain to synchronous domain, but also to align the data with a specific value of the global system counter. The overall system may be synchronous or asynchronous, but somewhere in the design there exists a global

synchronous free-running counter; it's purpose is to align events happening in the system to a specific time instance (determined by the clock rate). Such events can, for example, be packet broadcasts in the sensor network that relate packet's arrival or transmission to a network timer (counter) value that is used for packet processing.

We would like to focus on globally synchronous systems that use asynchronous circuits to perform part of the computation. Decision of using asynchronous circuits may be based on some of the advantages that asynchronous circuits have in power consumption, their robustness to delay variation and process variation, as described in the Introduction section of this thesis.

As stated earlier, such systems often have a requirement (in terms of counter value) on *when* the processed data should be "released" from the asynchronous circuitry to the rest of the system. In order to satisfy this requirement, we have created a highly efficient, specialized interface that performs above described functions. We have developed several modifications to the standard scheme, as well as used some of the previously implemented techniques for our design of this asynchronous-to-synchronous interface. Next section gives detailed description of our interface implementation and goes over it's operation.

An example of such a system would be a high-throughput router in a globally synchronous on-chip network of processing elements, as depicted in Figure 5.3. Due to the fact that in many such networks communication between the processing synchronous elements is rare and usually happens in bursts, a synchronous router would waste a lot of power and/or require complex clock gating. As the results of out previous evaluations show, asynchronous circuits are a perfect solution for such a task.

In this scenario, all of the processing elements are timed with a global clock.

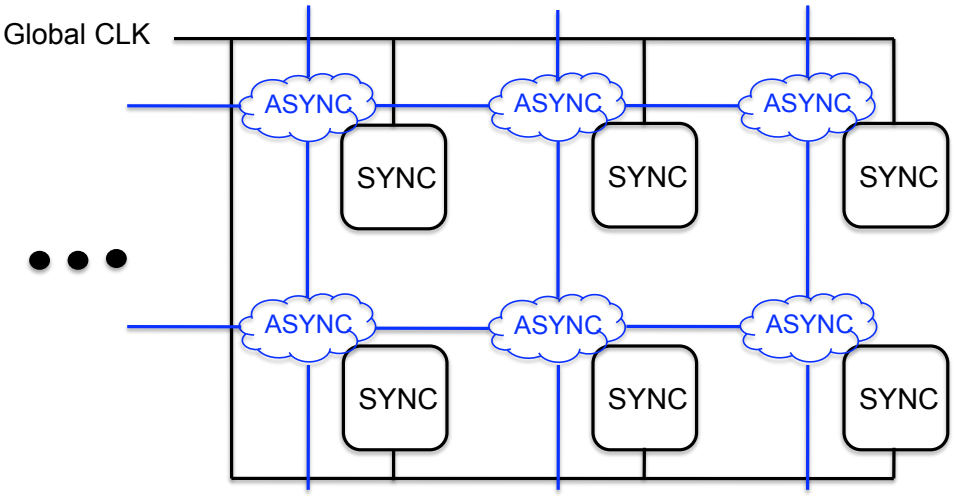

Figure 5.3: On-chip Globally Synchronous Network

Packets that are transmitted between these elements usually have to be delivered to a remote processing element at a specific clock 'tick'. The exact packet latency through the router is not known, and depends on various parameters, such as router congestion. However, the maximum delay through the router is defined. With this information we can use an asynchronous router and deliver the packet as soon as possible; then synchronize it with the global clock at the destination and wait until the proper global timer value to release the packet to the destination processing element.

5.3 Proposed Interface Overview

Since majority of the circuits that we design are QDI (quasi-delay insensitive) asynchronous circuits, we would like to focus on an interface that is data-driven and highly efficient in terms of throughput, latency and power consumption. This interface receives asynchronous data tokens and synchronizes them with a global system clock. The output of such an interface would be either valid synchronous data, or valid asynchronous data that has been aligned with a clock signal. Be-

sides aligning the received data with a clock signal, our interface release this data only at a specific value of the global timer. In case of synchronous output all the subsequent processing circuitry will be synchronous in nature. If the output is a *synchronized* asynchronous token, the subsequent circuitry will be asynchronous in nature; however, the timing of this token is well defined after synchronization. This information (alignment with a specific timer value) may be used to for correspondence with matching software (such as a discrete event simulator), or with a logically equivalent fully synchronous system.

The block diagram of such an interface is presented in Figure 5.4. As mentioned earlier, the input is an asynchronous QDI signal (data and enable/acknowledge). In order to provide the ability to have deterministic timing through the system we perform not only synchronization, but also alignment to the Synchronous Timer. Clock signal (CLK) is provided externally for data synchronization purpose. The output of the block is data, synchronized with the CLK signal. This data could be synchronous (single rail logic), or *aligned* asynchronous (multi rail with enable) to properly implement the QDI handshake.

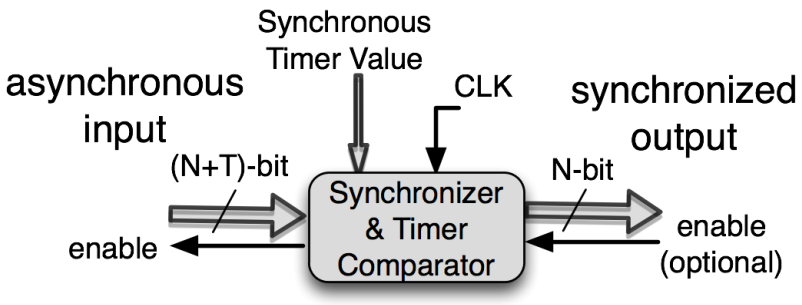

Figure 5.4: Synchronizer Overall Diagram

The output of the interface may have a different number of bits (N-bit) than the asynchronous input (N+T -bit), since the original asynchronous input to the interface must contain the time value, which indicates when the data has to be released after synchronization. This timer value in most occasions will not need to

be passed on from the interface to the succeeding circuit.

A common use for such an interface would be a system that requires globally deterministic operation, while locally making use of data-driven (asynchronous) behavior. The interface that we are discussing is able to track global time (by aligning data to the synchronous timer), and gives designer the flexibility to perform some processing in asynchronous domain. If all the subsequent circuitry is synchronous, we trivially modify the last asynchronous stage to output single rail data compatible with the rest of the synchronous system.

5.4 Proposed Interface Detailed Description

We would now like to discuss details of the interface proposed in the previous section and explain why it would be advantageous to use this interface at some of the asynchronous/synchronous boundaries in the modern systems.

In order to minimize the number of synchronizers required at the boundary, we use validity signals that indicate that all the data bits on a given channel are in legitimate (valid or neutral) states. For example, instead of an asynchronous eMx1of2<10> channel (10 dual-rail data values and one enable wire) we would use an evMx1of2<10> channel, which adds another value (wire) that needs to be transmitted between different circuit parts. The advantage of such an approach is that the input validity does not need to be calculated locally at the next circuit element, but is used from the output validity wire of the previous circuit element. The disadvantage is that there is one more wire per each circuit element, which complicates physical channel routing. However, if the channels are wide (multi-bit), the overhead of one more wire is negligible. In our opinion this cost of an additional wire is amortized by the reduction in circuit's area and power consumption [10].

The validity value is calculated by completion trees (C-tree). A C-tree checks

for the presence of a valid data (or neutral data) over all the bits within a channel by using C-element circuit primitives [33].

With the validity-based channels, instead of synchronizing each data bit with the clock, we only need to synchronize the validity. According to the QDI principals, computation of validity is constructed in such a way that the validity is the last value that changes on a channel (*all* data values must be in an allowed state before validity's value changes). We, thus, note that once the validity is synchronized with the clock, we are guaranteed that all the data bits are synchronized as well.

The detailed description of our synchronizer is shown in Figure 5.5.

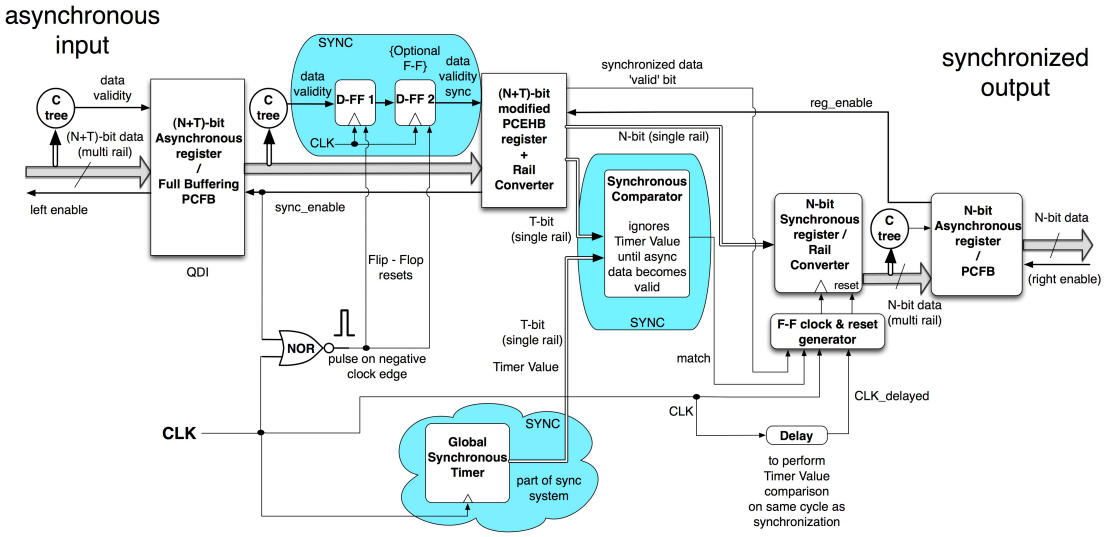

Figure 5.5: Synchronizer Detailed Diagram

PCFB and PCHB (PCEHB) templates used in this description stand for Pre-Charge Full Buffer and Pre-Charge Half Buffer (Pre-Charge Enable Half Buffer) respectively. Two half buffer structures are required to hold one complete data token; alternatively, only one full buffer could be used in their place. Few advantages of a half buffer, however, is a smaller number of transistors, and a smaller number of transitions per cycle than a full buffer. Half buffers are especially useful

in designs, where full *slack* is not required [20]. Both of these templates and their various implementations have been thoroughly studied by Lines and Fang [10, 18].

5.4.1 Input Stage of the Synchronizer

All the C-tree blocks are used to produce the channel's validity signal, as discussed earlier. The *asynchronous input* is received by the interface from the left side. The interface structure is independent of the width of input data and is denoted as an (N+T)-bit value for reference. In order to fully decouple the interface from the preceding asynchronous circuitry, we store the data token in a full buffer, PCFB-style. PCFB immediately releases the left handshake, before the data token is passed to the next (right-side) element of the interface. As a result, the preceding (left-side) asynchronous circuitry may start processing the next data token before PCFB completes the right part of the handshake.

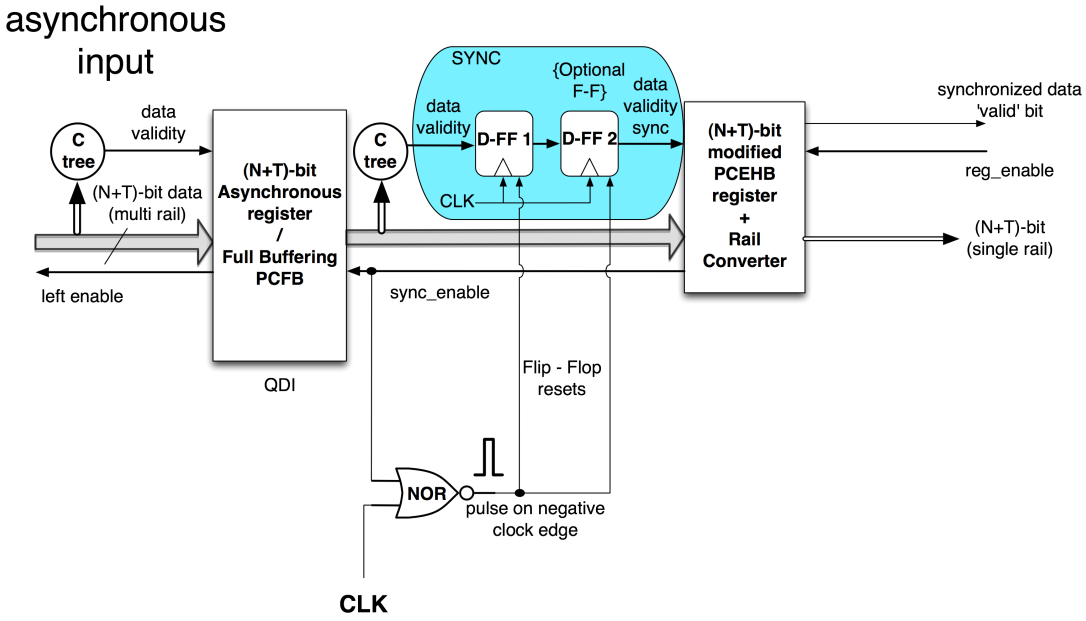

Figure 5.6: Input Stage of the Synchronizer

The synchronization with a clock (CLK) happens between the PCFB and

PCEHB elements as shown in Figure 5.6. The validity of this channel is passed through flip-flops, as discussed earlier. If designer has prior knowledge of the correlations between the time of arrival of asynchronous tokens and the CLK signal, only one flip-flop may be used in some scenarios. Example of such case would be if the asynchronous token always arrives before the CLK and provides enough time to satisfy the flip-flop's setup and hold times. Our structural design of the flip-flops will be discussed in the next subsection.

However, if nothing is known about the clock's relationship to the arrival time of asynchronous tokens, the standard solution of two sequential flip-flops may be used. The output of the second flip-flop is now represented by a *data validity sync* signal that is aligned with the clock. In this case, the enable coming back from the PCEHB *sync_enable* to the PCFB also happens only after the positive edge of the clock and is thus aligned with CLK signal as well. The enable actually gets asserted almost right after the positive clock edge where the synchronization has completed (plus a small delay from the transitions coming from the control part of the PCEHB).

In a regular PCEHB buffer reshuffling, the data at the output of the buffer appears two transitions after a valid data has arrived at the input of the buffer. However, in our case, we cannot allow that two happen, because that would mean that since only validity is synchronized, the data would appear at the output of PCEHB before the synchronizations occurs and will cause the later part of the interface and the succeeding circuitry to malfunction. In order to avoid this scenario, we modify the evaluation stacks for the data rails of the PCEHB and add one transistor to pull-up stack and one transistor to pull-down stack to check for the synchronized channel validity value (Lv) as demonstrated in Figure 5.7. This modification has to occur at stacks of all the data bits. It guarantees that

the output data will become valid only after the validity gets synchronized with the clock. Since we now check for left validity on the data evaluation rails of the PCEHB, we don't need to check it again in the control part of the PCEHB. Thus, the control gets simplified, as depicted in the figure.

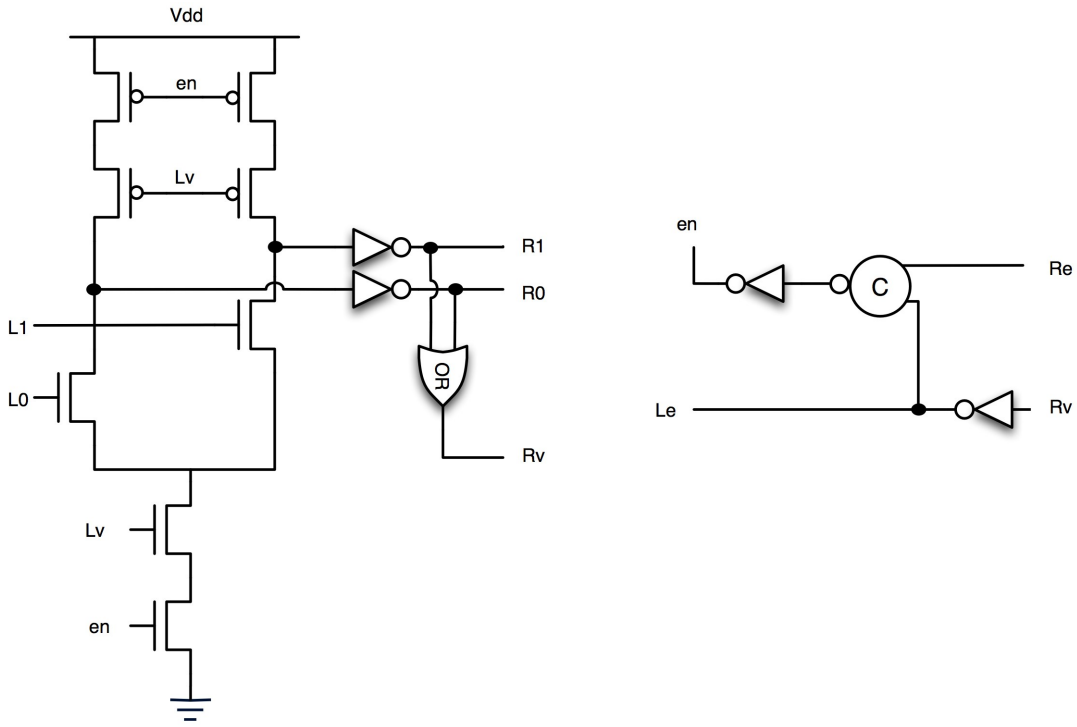

Figure 5.7: Modified PCEHB Element

In this implementation, since both senses of the left validity are checked at the data rails, the right validity implies the left validity; thus the left enable is produce merely by inverting the right validity. Other than this small modification to both data and control evaluation circuitry, the buffer behaves similarly to the original PCEHB implementation, described by Fang [10].

The reason to have a PCEHB stage as the second set of buffers instead of PCFB is analyzed next. In case of a second PCFB stage, there is a chance that there is a complete data token residing in the second hypothetical PCFB (the left part of the handshake resets right after the token is received, regardless of whether the token

has been processed on the right side of the PCFB). There could be a scenario when this synchronized token is not immediately processed by the synchronous circuitry; for example if the the result of the comparison of the timer value with the newly arrived token T-value indicates that the data needs to be held until the appropriate timer 'tick' happens. In such a case, if another data token arrives from the left (from the first PCFB stage), the validity gets synchronized and aligned with the clock edge, but the data doesn't get stored in the PCFB yet, because the place is occupied by the previous token. Logically, this is allowed, but the problem is that the new token gets held up at the PCFB's input and will get out of sync with the clock edge eventually. If so, it can lead to a metastability scenario in the succeeding synchronous part once it enters the synchronous comparator, because the data value will not be tightly related to the clock edge any more. The simplest solution to this problem is not to finish the left handshake, before the synchronous circuitry processes the previous token. To do that, one could use a different handshake reshuffling that interleaves the left and the right side of an asynchronous handshake. A PCHB (PCEHB) template exactly follows this behavior and is used as a second stage in our design for that reason.

5.4.2 Flip-Flop Implementation Details

In the common two flip-flop synchronization scheme, the number of cycles that it takes to synchronize a signal is variable. That number is dependent on the time of data arrival in relation to the *next* clock edge. This synchronization will be on average around two cycle on the valid part of the 4-phase handshake and two cycles on the neutral part of the handshake, since both senses of the validity are synchronized. This seems like a waste to us, because the neutral phase of the validity doesn't really need to be synchronized, since no computation is performed

(no new data is introduced), as long as further processing within the interface is implemented in such a way that the interface operations is not broken. So in order to minimize the number of clock cycles it takes to fully synchronize the valid data token, we don't synchronize the neutral state of the validity. Since flip-flops are state-storing elements, we need to clear that state accordingly to mimic the progression of the neutral part of the handshake at the flip-flops' outputs. For that we use a special asynchronous reset.

Alternatively, instead of imitating the neural phase in the four-phase handshake, one could choose to use a two-phase handshake instead. In that case any change of the request signal (validity signal in our case) would be synchronized, however, every change of that signal would indicate the arrival of the new set of data. The concern with the two-phase handshake based circuits, though, is their complexity. The surrounding circuitry always ends up being bigger (in terms of area) and more difficult to design than in the four-phase handshake case. For that reason we decided to stay with a four-phase handshake based design and to implement a special asynchronous reset on the flip-flops' outputs. Analysis, conversion and comparison of two- and four-phase handshake based circuits are presented by Nowick [25].

The forward (valid token) part of the handshake is synchronized through the flops, as originally intended. However, after the PCEHB element accepts the data token, we clear the flip-flop state and reset all intermediate signals to the neutral state of the handshake, thus, mimicking the advancement of a neutral token through the synchronization circuitry.

The flip-flops are reset with a short pulse generated with a NOR gate that combines the *sync_enable* signal, indicating that the token has been accepted by the succeeding PCEHB and the negative edge of the CLK signal (to provide a

safety margin and avoid collision with the next data token). One thing to note, is that since we are using the modified version of the PCEHB, the *sync_enable* signal becomes active only after the data token is passed on from the PCEHB to the Synchronous Comparator. Such reset scheme minimizes the number of CLK cycles required for synchronization and provides a safety margin for the flip-flop reset.

The length of the negative voltage pulse (assuming a clock rate that is longer than an asynchronous handshake completion time) is determined by how long the *sync_enable* stays active. Since both of the flip-flops are reset simultaneously, a designer should use caution while laying out the reset signals and make sure that the reset to the DFF1 is not longer (in terms of wire length) than the DFF2 reset wire. This restriction occurs due to the fact that the neutral output of DFF1 is not directly checked by any circuit and we assume that DFF1 has reset by the time *sync_enable* changes it sense. This assumption is safe, since there are several transitions from the time data validity sync goes to state '0' to the time *sync_enable* goes to neutral state (lengths of these transitions also determine the length of the reset pulse).

We have studied various flip-flop implementations and have selected a modified version of the C^2MOS (Clocked Complementary Metal Oxide Semiconductor) flop implementation. Such design minimizes the impact of races and also reduces the clock slew rate requirement. C^2MOS is fast, energy efficient and small which perfectly fits into our interface requirements. We augment the commonly used implementation with either full combinational feedback, discussed by Manohar [20]; or with conditional feedback (pseudo-static behavior) as explained later. The details of our flip-flop design are presented in Figures 5.8 and 5.9.

The difference between the two flip-flop variations is the addition of one more

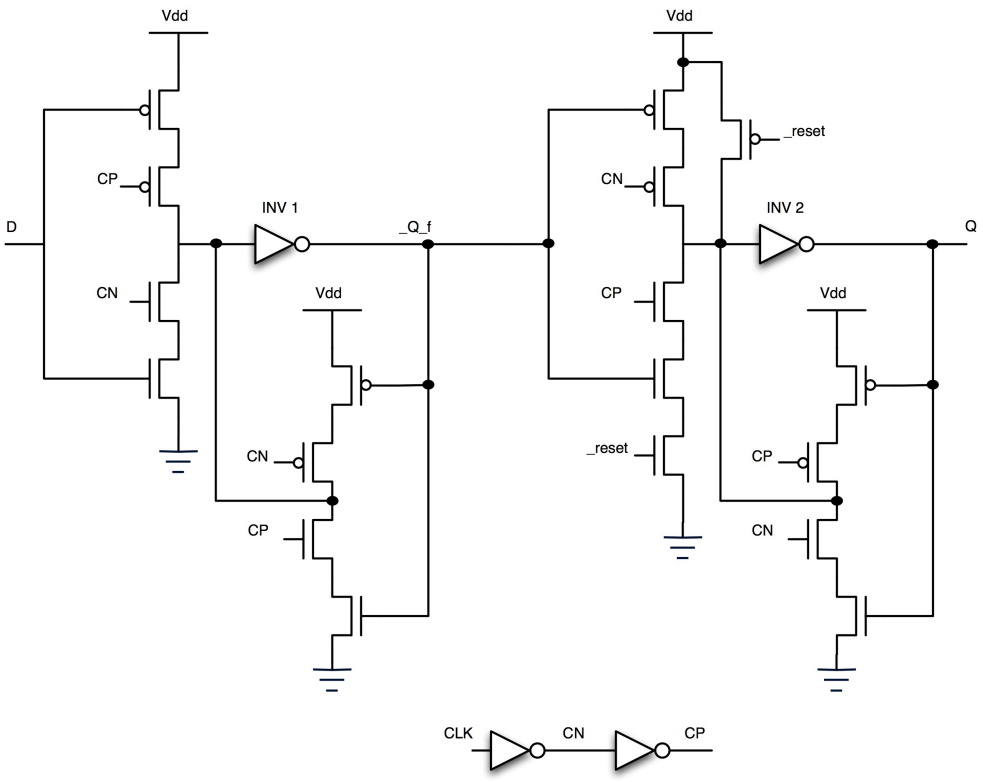

Figure 5.8: C^2MOS flip-flop with conditional feedback

transistor in each stack in the feedback networks. The advantage of the fully combinational feedback approach is that the intermediate nodes are always driven, regardless of the state of the clock, and regardless of the delays of the clock inverters and their corresponding wires. However, that comes at the cost of a few more transistors and higher power consumption (short circuit current) when the feedback's state needs to be flipped. Also the combinational feedback inverter needs to be carefully sized to make sure that the main stack can overpower the feedback when changing the stored value.

The approach with conditional feedback is thoroughly analyzed by Yeo [30] and Brodersen [22]. All of the tradeoffs and power analysis are presented, as well as several other flip-flop implementations.

Besides our asynchronous parallel and serial chip resets (not depicted in the fig-

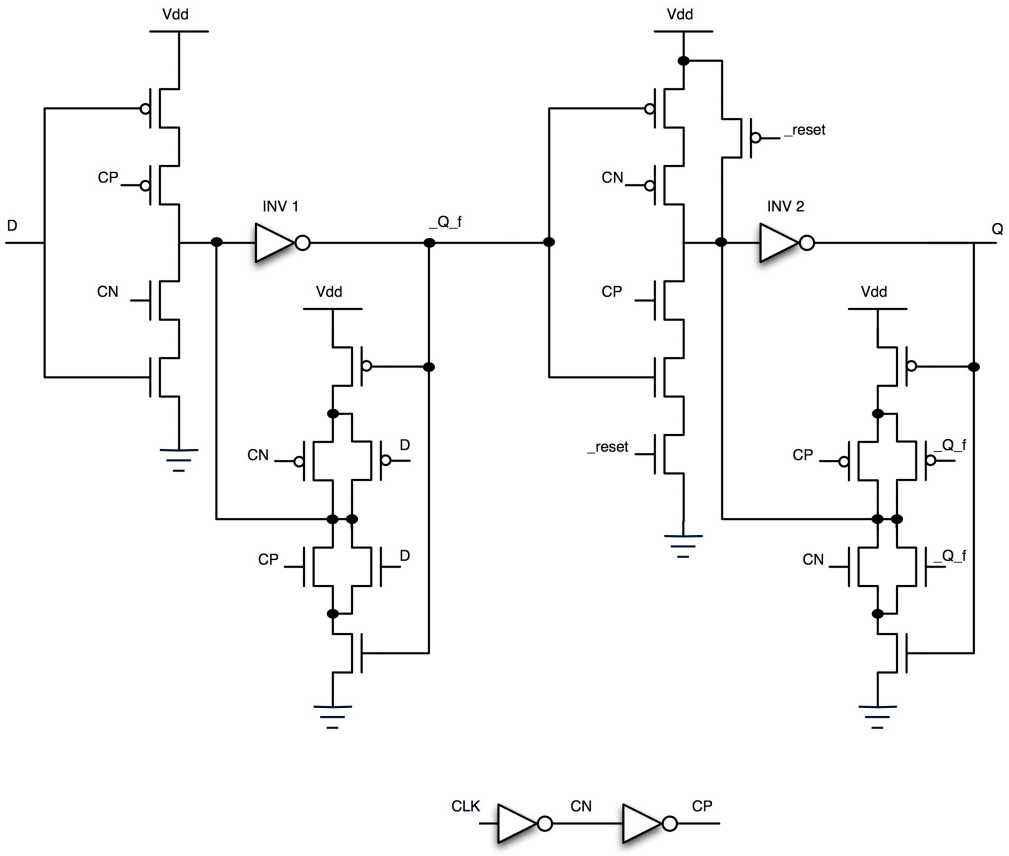

Figure 5.9: C^{2}MOS flip-flop with full combinational feedback

ures), we have added an asynchronous *_reset* signal to the second latch of each flip-flop. We generate this signal internally as described previously. These *_reset* inputs are used to perform a fast reset at the neutral phase of the asynchronous handshake. The "forced" reset is only performed at the second latch. The first latch is passively overwritten during the negative clock cycle, as soon as pre-synchronized validity value changes to zero, while it is transparent. So after the negative clock edge and once the *sync_enable* value becomes active, we write the first latch of each flop to zero through it's D-input, and force the second latch of each flop to a zero state using a parallel reset transistor and a series cut_off transistor (depicted on Figures 5.8 and 5.9). At the moment of the forced write, the second latch is non-transparent and no glitches can propagate forward from the preceding latch.

5.4.3 Synchronous Circuitry

The N-bit PCEHB buffer also works as a rail converter. The value comes out of this buffer in a single rail format, as opposed to a multi rail encoded format in the previous asynchronous stages. At that point, our token looks completely synchronous. The acquired token contains two sets of information: the Data Release Time Value (DRTV) that indicates when the data is supposed to be passed to next stage; and the actual data that needs to be passed on.

As shown in Figure 5.10, the Data Release Time Value is then compared to the value of the Global Synchronous Timer. The Global Timer is controlled by the same CLK signal as the synchronization flip-flops of the previous stage; thus, they are exactly in sync with each other. The decision of whether to release the arrived data to the next stage depends on the result of the comparison of the DRTV and the Timer Value.

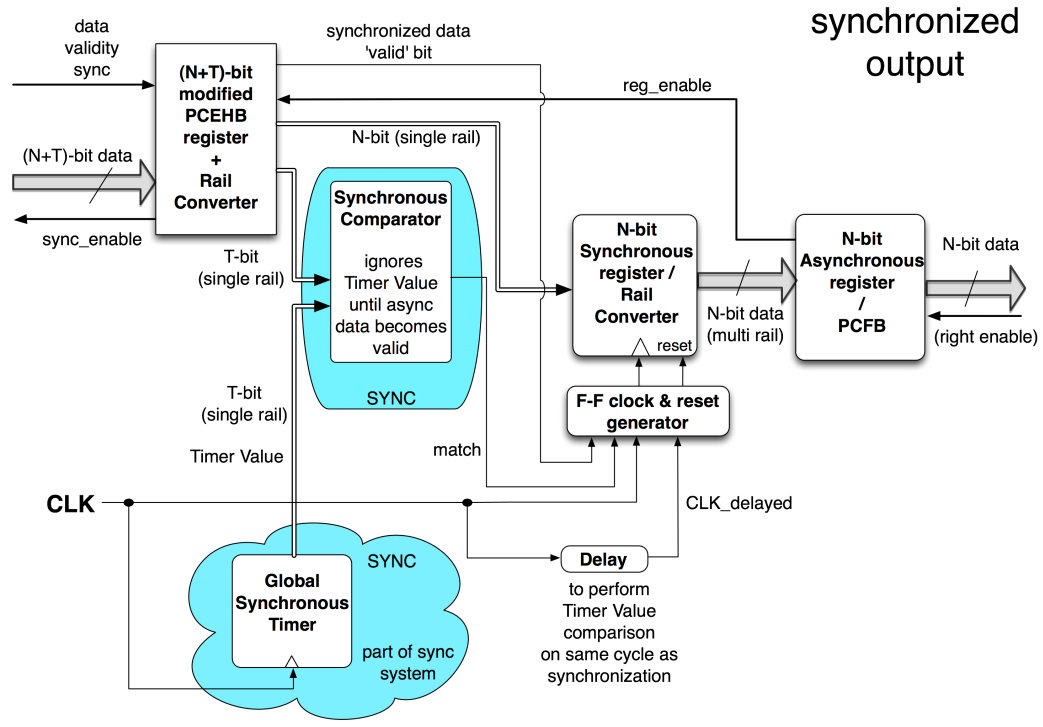

Figure 5.10: Synchronous Comparator and Ouput Stage of the Synchronizer

To make our interface more robust, we put a restriction that, if desired, all the communication with the (N+T)-bit PCEHB asynchronous stage has to be able to finish within the same clock cycle as when the (N+T)-bit data was received by the Synchronous Comparator. This would guarantee maximal throughput of our interface and ability of processing the arrived token (through the entire interface) within one clock cycle in the best scenario (if one flip-flop is used for synchronization, and the data arrives within an allowed time window of the negative clock cycle). The requirement of one clock cycle processing is non-trivial; that is why we would like to focus on this scenario.

In order to allow synchronous comparison on the same clock cycle as synchronization, we delay the clock arrival to the N-bit Synchronous register by the time it takes the signal to travel from the output of DFF2 to the N-bit Synchronous register (including the DRTV comparison with the Timer Value). This would allow the timing comparison function and all the synchronization to remain transparent to the rest of the design. The exact clock delay is bounded by time it take the signal to propagate from the positive edge of the CLK signal (which is when *data validity sync* goes up) to the time instance when $\_Q\_f$ of the flops at the N-bit Synchronous register goes up, as we will describe in the next subsection. For a point of reference, for Synchronous Comparator working as a simple XNOR-based 5-bit comparator in IBM's 45 nm technology node, this delay is on the order of 400 ps, which is incomparably faster than common contemporary clock rates. Such delay corresponds to approximately 9 slow-fast inverter pairs (chain of small inverter - large inverter pairs) to implement the delayed version of the CLK signal. The delayed version of the global clock signal is called *CLK_delayed* in our diagrams.

The delayed clock technique, applied in our interface, is similar to a *time-borrowing* approach used by synchronous logic designers [14].

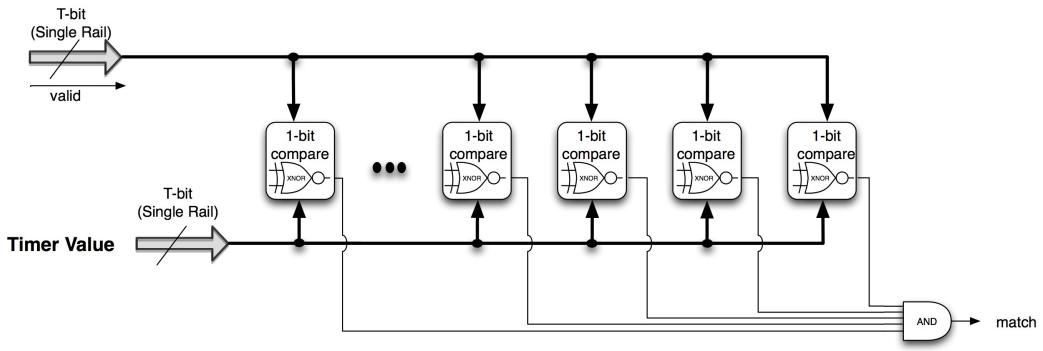

Figure 5.11: A Variant of Synchronous Comparator Implementation

A possible conceptual implementation of the Synchronous Comparator is presented in Figure 5.11. Such a design consists of T parallel 1-bit comparators (for example implemented as XNOR gates) and a gate (or set of gates) that combines the result of these 1-bit comparisons. In case of XNOR-s, the gate that combines the outputs is an AND gate (or a tree that implements the AND function if the fan-in is large). The output of the AND gate is indicated by a *match* signal that is in a 'true' state if all comparators indicate that their DRTV bit matches the corresponding Global Timer bit, and is in a 'false' state otherwise.

The AND gate that generates the *match* signal in Figure 5.11 does not create any potential glitch problems for the succeeding asynchronous circuitry (glitches can lead to deadlock in QDI-type circuits). Glitches can occur at the output of the AND gate, however, that can only happen during the negative phase of the *CLK_delayed* signal. Evidently, by construction of the delay element, we require that the output of the AND gate settles to a proper value before the positive edge of the *CLK_delayed* occurs. Thus, no glitches can propagate to the second set of latches in the flip-flops of the N-bit Synchronous register; all glitches are stopped at the first set of latches.

5.4.4 Synchronous Register Implementation

As long as the DRTV does not match the Global Timer, the result of the comparison is 'false' and the data does not get latched in the N-bit Synchronous register. In order to achieve that, we simply gate the clock input of the register's flip-flops with the resulting *match* signal value (and a few other signals, as discussed later on) of the Synchronous Comparator.

Once the Synchronous Comparator block finishes the computation (within the positive phase of the clock cycle) and the resulting value is 'true', the single rail N-bit data is passed to the Synchronous Register.

As mentioned earlier, we have restrained the interface to the single cycle synchronization requirement. In order to perform synchronization within one clock cycle from the time the data is outputted by the PCEHB, we modify the Synchronous register's clock and reset control signals. For the register itself we use the same C^2MOS type flip-flop as in the validity synchronizer, described in the earlier section. However, the control signals are generated differently and require some explanation.

The PCEHB element besides outputting the data to the Synchronous Comparator, generates a *valid* bit that indicates that all the data bits are in the valid phase. The flip-flops in the Synchronous register are activated (clock is enabled) only when this *valid* signal is true. This means that all the changes of the Timer Values are effectively ignored until the *valid* signal indicates that (N+T)-bit token is available for time value comparison. In order to implement this type of flip-flop clock control, we combine the *valid* and *CLK_delayed* signal (and the *match* signal described previously) with an AND gate, as indicated in Figure 5.12.

The same *valid* signal can be safely used to power-gate all synchronous circuit elements and the successive circuitry to minimize the standby power consumption.

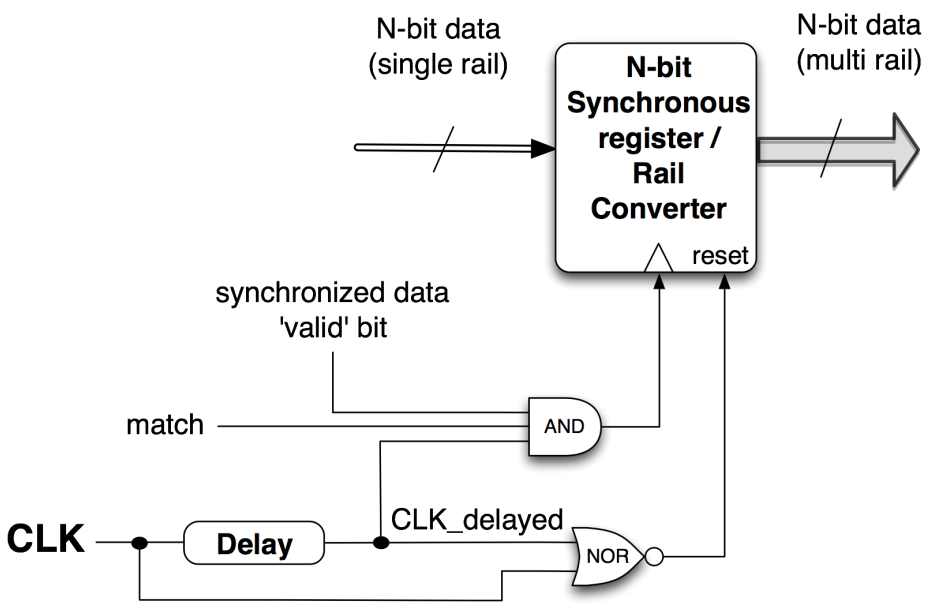

Figure 5.12: Synchronous Register / Flip-Flop Control Signals

Unlike the AND gate of the Synchronous Comparator, *no* glitches can possibly occur at the output of the three-input AND gate that controls the clock input of the flip-flops in the N-bit Synchronous register in Figure 5.12. The reason for that is the AND gate's input arrival sequence. For the up-going transition of the clock input of the Synchronous register, the only possible input sequence is *valid* goes high, then *match* goes high (after the CLK edge that resulted in a positive result of the timing value comparison), and last *CLK_delayed* goes up. These values all change sequentially, monotonically and are guaranteed to stay in their states, until the *valid* bit goes down, which by construction has to happen before the next CLK tick arrives. On the down-going transition of the clock input of the Synchronous register, the *valid* bit going low blocks all further triggers from *CLK_delayed*. The next time *valid* bit can go high is only on the next CLK cycle (if the validity synchronization occurs immediately on the next cycle, and only one flip-flop is used for the validity synchronization), however, during that time the clock input of the Synchronous register is blocked by the negative value of the

CLK_delayed. By the time the positive edge of *CLK_delayed* arrives, the *match* signal is guaranteed to already be in the correct state (again by the construction of the delay unit), and will either allow the *CLK_delayed* edge to go through to the Synchronous register, or block it's edge until the timing value comparison results in a positive outcome. Thus, all the transitions are monotonic and no violations of the QDI protocols can occur.

Due to the requirement of the same cycle data output from the N-bit Synchronous register and the constraint that we have to be able to receive the next valid data token on the next cycle (in case we use single flip-flop for the validity synchronization), we have to force the reset of the synchronous flip-flops in the Synchronous register. This guarantees that by the time the next clock cycle comes, the flip-flops are ready to receive another set of data. In order to do that, we combine the CLK signal and *CLK_delayed* signal with a NOR gate. Such combination produces a pulse signal starting at the positive edge of the *CLK_delayed* signal. The length of the reset pulse is determined by how much the CLK signal is ahead of the *CLK_delayed* signal. In most typical applications (where the clock is not extremely aggressive), the amount of the clock delay will be small, compared to the cycle time. In such cases, the reset pulse will last for the majority of the negative CLK phase. The reset pulse terminates once the CLK signal goes back up to the positive clock phase, thus producing a guard band around the positive edge of the *CLK_delayed* signal.

Such flip-flop reset behavior is allowed, since we only actively force reset of the second latch, which is not transparent during the negative clock phase. The output value of the first latch remains untouched by the reset signal and will settle to the zero value during the negative clock phase, if there are no pending tokens. Again no glitches can propagate forward, since the second latch is non-transparent in the

negative clock-phase and, thus, can only either stay at a zero-value (if the data is in neutral state), or have a monotonic one-to-zero transition once the reset is applied. This type of reset guarantees that the signal integrity is well maintained at all the latch outputs.

The Synchronous register also works as rail converter and generates multi rail data signals out of single rails. For example, if dual-rail signaling is used, the single rail is passed on directly as the true rail of the encoded data signal; and in parallel, it is also passed through an inverter to generate the false rail.

5.4.5 Synchronizer Output Stage

From the output of the N-bit Synchronous register, valid data goes to the N-bit Asynchronous register, as depicted in Figure 5.10. This register is implemented as a full buffer stage (PCFB), to decouple the Synchronous register and the next data processing block, outside of the interface (not shown). The validity signal of this data is computed in parallel by using a C-tree. Once all the data is in the allowed state (indicated by the validity), the handshake with the (N+T)-bit modified PCEHB register is completed, and the PCEHB is ready to receive a new synchronized valid token from the leftmost PCFB. All of this will happen within one cycle, as long as the timing comparison had a matching outcome.

If the two flip-flop synchronizer scheme is used and data tokens are backed up at the left side of the interface (tokens are waiting to be synchronized), the data is able to be accepted every other cycle at maximum (assuming matching time values and same cycle output from the right side of the interface), which corresponds to the maximum throughput of this interface for the two flip-flop synchronization. The data could potential be accepted, synchronized and processed every cycle in this interface, if one flip-flop synchronizations scheme is used, however, that comes

with a higher probability of metastability propagation at the synchronizer, which in most cases is not desirable behavior.

The N-bit Asynchronous register / PCFB at the right side of the interface performs a triangular handshake and guarantees that once the *reg_enable* is asserted, the data is successfully latched by the PCFB elements and that the Synchronous Comparator has successfully finished comparing or halting the data. PCFB reshuffling allows the next token to start processing before the previous data token gets retrieved by the circuitry succeeding the interface. The output part of the Asynchronous register should be implemented according to the data format that is required by the succeeding circuitry. If asynchronous processing is employed afterwards, the next element will perform a regular 4-phase handshake with the N-bit Asynchronous register / PCFB.

In case all further processing (after the interface) is synchronous, the output should be converted back to single rail data. If dual rail encoding was utilized, we would drop the false rail and use the true rail as single rail data. At the time the data is passed from the last PCFB stage to the succeeding element, it is still aligned with the global *CLK_delayed* signal. Part of the next clock cycle has already been taken up by the propagation delay through the Asynchronous register, and that delay has to be taken into account during the synchronous timing simulations. If synchronous processing is employed afterwards, the *right enable* signal is not part of the interface and can be controlled by one of the local signals, for example by the clock edges, which would provide one clock phase for synchronous processing and second phase for acknowledging the neutral phase. In case of dual rail encoding, the all 0-s output form the Asynchronous register does not correspond to a valid data token and should be treated by the succeeding synchronous circuitry accordingly.

Chapter 6

Additional Related Research

Various CAD tools exist for VLSI synthesis using standard cells such as the industrial synthesis tool offerings from Cadence and Synopsys. Standard tools, such as Synopsys Design Compiler [4], mainly use synchronous static CMOS standard cell synthesis. These tools can take a high-level Verilog/VHDL description and synthesize it into synchronous gate-level netlist using a supplied library of standard cells. Since most of the standard cell libraries do not supply transistor-level descriptions, the flexibility of performing transistor-level optimizations is taken away from the designer.

Some work has been done in academia to extend/replace the standard industrial tool flow to allow circuit family-specific synthesis. Several researchers have achieved substantial improvements of an optimized metric (area/power/throughput) using asynchronous circuit implementation and synthesis.

A. Martin, *et al.* showed general techniques for compiling asynchronous circuits from a high-level description language CHP into transistor-level designs using data-driven decomposition such as control-data and projection [39].

Farhoodfar, *et al.* illustrated asynchronous circuit synthesis from the high-

level description language, Communicating Sequential Processes (CSP) [37], into a library of asynchronous standard cells based on Pre-Charge Half Buffer (PCHB) and Pre-Charge Full-Buffer (PCFB) templates [15].

P. Beerel, *et al.* demonstrated a back-end design flow for asynchronous standard cells that allows low-level (transistor-level) circuits to be implemented from a schematic description using a single-track full buffer (STFB) template [12]. Bereel and his colleagues have also released an asynchronous standard cell library, the High-Speed Asynchronous Pipeline Cell Library, that is available for public use [11].

Ellervee, *et al.* described their techniques for automatic synthesis of asynchronous circuits for RTL-based design descriptions [28]. While both single- and dual-rail asynchronous circuits were considered, no transistor-level analysis was shown.

To the current knowledge of the authors, no general tools or techniques for compilation of an industry accepted high-level description language (Verilog or VHDL) into synchronous and asynchronous transistor-level circuit families have been previously demonstrated using a *unified toolflow*.

Chapter 7

Suggestions for Further Improvements

The Toolflow Evaluation section demonstrated that our proposed toolflow is efficient for both circuit families under consideration, depending on the applications and targeted metrics. The design cycle time is drastically improved, if our techniques are used.

As for further improvements to this work, we suggest to focus on the synchronous-to-asynchronous conversion algorithm. Improvements in the algorithm may lead to significant benefits in terms of asynchronous circuit power consumption and occupied area.

A good starting point to improve the conversion would be to add an option of having the ability to control the "effective pipeline depth" of the asynchronous circuit. This can be done by combining several consecutive CMOS standard cells into one evaluation stack of the asynchronous template. This technique will radically decrease the effective transistor overhead per asynchronous standard cell.

Depending on the complexity of the standard cells under consideration, a different number of standard cells should be used per asynchronous template. The optimal number may be determined through simulation in each given design.

The previously described technique does not involve any complex toolflow modifications or additions. In case the obtained power savings are not sufficient, we advise to create a compiler to perform further asynchronous specific optimizations. As an example, such compiler could take a high level circuit description and transform it into efficient asynchronous Verilog-type primitives still at high level. A compiler could also operate on the asynchronous gate-level netlist obtained after running Verilog-to-ACT tool. This compiler should be able to read asynchronous Verilog-level description of the design and/or an asynchronous gate-level ACT type netlist.

The asynchronous circuit compiler can perform various types of optimizations to further improve the power consumption. A few things worth mentioning for such compiler design are the following. The number of copy-elements should be minimized, thus the number of high-fanout out nodes should be reduced by the compiler. This can often be done by means of using asynchronous splits and merges throughout the computation. The compiler can also perform control-data decomposition [20]; busses should be combined into separate instances with control actions performed only once per entire bus, instead of once per each bit. Instead of using highly pipelined templates, such as PCHB and PCFB, function blocks and asynchronous register 'read' and 'write' ports may be used to reduce transistor overhead. Some additional techniques that may be used are described by Manohar [20], Martin [23, 39] and Teifel [35].

It is also important to evaluate other asynchronous families to further understand the synchronous / asynchronous tradeoffs. The templates that we advise to

consider are: PCHB (pre-charge half buffer without validity forwarding) for higher throughput, WCHB (weak-condition half buffer, where computation is performed on both p- and n- stacks) for smaller number of gates [20], HCHB (half cycle half buffer that doesn't check the neutrality phase) for lower power consumption [16], STFB (single track full buffer with reduced transistor count) for lower power consumption [12].

Chapter 8

Conclusion

In this thesis we have proposed a novel way of designing complex VLSI circuits. We have presented a new hybrid synchronous / asynchronous toolflow that drastically reduces the design time for VLSI projects. The presented approach is especially useful in mixed-circuit type medium-scale projects, where accurate measurements early in the design cycle can drive many high-level architectural decisions. Using our methodology, the designer can obtain accurate transistor-level measurements, including throughput, power, and area, at the initial stages of design development.

Our toolflow allows designers to select the ideal circuit family for each digital block based on the original design metrics, without necessitating thorough expertise in all logic families. Furthermore, we bypass the cumbersome layout place and route step until the final stage of the design cycle, which greatly accelerates the design phase.

We also present an innovative tool, called COSIM that interfaces various high-level and low-level simulators (both synchronous and asynchronous). This tool allows the designers to perform mixed-circuit simulations at various levels of abstraction. COSIM also has the capability of automatically generating proper test-

ing environments for both synchronous and asynchronous circuit implementations.

Based on the evaluation results of our toolflow, we conclude that depending on the application, there are scenarios were it is beneficial to use synchronous circuits, and other scenarios were asynchronous circuits provide better results. We provide guidelines that help designers determine which type of circuit family to consider, based on the high-level structure of their designs and on the metrics that designers are striving for.

Lastly, we present a highly efficient asynchronous-to-synchronous interface for applications that require deterministic behavior of the global system. This interface allows partial implementation of the design in an asynchronous manner without loosing exact event sequencing in terms of clock cycles. Such system behavior can easily be modeled in a software discrete event simulator which will have an exact correspondence with the actual hardware system.

Bibliography

[1] Nangate 45nm open cell library. In *http://www.nangate.com/*, 2009.

[2] ITC-99 benchmark homepage from University of Texas. In *http://www.cerc.utexas.edu/itc99-benchmarks/bench.html*, 2010.

[3] Overview of ITC-99 benchmarks form Torino, Italy. In *http://www.cad.polito.it/downloads/tools/itc99.html/*, 2010.

[4] Synopsys tool references, simulation manuals, tool executables. In *https://solvnet.synopsys.com/*, 2010.

[5] F. Akopyan, R. Manohar, and A. B. Apsel. A level-crossing flash asynchronous analog-to-digital converter. In *Proc. of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 11–22, 2006.

[6] R.E. Bryant and et al. Limitations and challenges of computer-aided design technology for CMOS VLSI. In *Proc. of the IEEE*, pages 341–365, 2002.

[7] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. In *IEEE Transactions on Computers*, pages 421–422, 1973.

[8] D. M. Chapiro. *Globally-Asynchronous, Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[9] D. Fang, S. Peng, C. LaFrieda, and R. Manohar. A three-tier asynchronous FPGA. In *International VLSI/ULSI Multilevel Interconnection Conference*, 2006.

[10] David Fang. Width-adaptive and non-uniform access asynchronous register files. Master's thesis, Cornell University, December 2003.

[11] M. Ferretti and P. A. Beerel. USC's PCHB based asynchronous gate library. In *http://jungfrau.usc.edu/new/research/current/last/index.html*, 2004.

[12] M. Ferretti, R. O. Ozdag, and P. A. Beerel. High performance asynchronous ASIC back-end design flow using single-track full-buffer standard cells. In *Proc. 10th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 95–105, 2004.

[13] R. Ginosar. Synchronization and arbitration. In *ACiD Summer School on Asynchronous Circuit Design*, Grenoble, France, July 15-19 2002.

[14] G. Jung, V. Perepelitsa, and G.E. Sobelman. Time borrowing in high-speed functional units using skew-tolerant domino circuits. In *Proc. of the 2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, page 641, 2000.

[15] H. Pedram K. Saleh, M. Naderi, M. H. Shafiabadi, H. Kalantari, and A. Farhoodfar. Synthesis tool for asynchronous circuits based on PCFB and PCHB. In *Proc. 9th Annual Computer Society of Iran Computer Conference*, 2004.

[16] C. LaFrieda and R. Manohar. Reducing power consumption with relaxed quasi delay-insensitive circuits. In *Proc. of 15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 217–226, 2009.

[17] W. Lim. Design methodology for stoppable clock systems. In *IEEE Proc. on Computers and Digital Techniques*, pages 65–72, 1986.

[18] Andrew Matthew Lines. Pipelined asynchronous circuits. Technical report, Caltech, 1998.

[19] R. Manohar, M. Nyström, and A. Martin. Arbiters are forever. Discussion and analysis of various arbiter circuits, 1995.

[20] Rajit Manohar. Asynchronous VLSI systems. Class Notes for ECE 574 at Cornell University, 1999.

[21] L. R. Marino. The effect of asynchronous inputs on sequential network reliability. In *IEEE Transactions on Computers, C-26*, pages 1082–1090, 1977.

[22] D. Markovic, B. Nikolic, and R. W. Brodersen. Analysis and design of low-energy flip-flops. In *International Symposium on Low Power Electronics and Design*, pages 52–55, 2001.

[23] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4), 1986.

[24] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *ARVLSI*, pages 263–278. MIT Press, 1990.

[25] A. Mitra, W.F. McLaughlin, and S.M. Nowick. Efficient asynchronous protocol converters for two-phase delay-insensitive global communication. In *Proc. of the 13th IEEE International Symposium on Asynchronous Circuits and Systems*, page 186, 2007.

[26] S. W. Moore, G. S. Taylor, P. A. Cunningham, R. D. Mullins, and P. Robinson. Using stoppable clocks to safely interface asynchronous and synchronous subsystems, 2000.

[27] M. Nystrom and A. J. Martin. Crossing the synchronous-asynchronous divide. In *Workshop on Complexity-Effective Design*, 2002.

[28] J. Oberg, J. Plosila, and P. Ellervee. Automatic synthesis of asynchronous circuits from synchronous RTL descriptions. In *23rd NORCHIP Conference*, pages 200–205, 2005.

[29] M. Pechoucek. Anomalous response times of input synchronizers. In *IEEE Transactions on Computers, C-25*, pages 133–139, 1976.

[30] M.-W. Phyu, W.-L. Goh, and K.-S. Yeo. Low-power/high-performance explicit-pulsed flip-flop using static latch and dynamic pulse generator. In *IEE Proceedings on Circuits, Devices and Systems*, pages 253–260, 2006.

[31] F. Rosenberger and T. J. Chaney. Flip-flop resolving time test circuit. In *IEEE Journal of Solid-state circuits, SC-17*, pages 731–738, 1982.

[32] J.N. Seizovic. Pipeline synchronization. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96, November 1994.

[33] M. Shams, J. C. Ebergen, and M.I. Elmasry. A comparison of CMOS implementations of an asynchronous circuits primitive: the C-element. In *International Symposium on Low Power Electronics and Design*, pages 93–96, 1996.

[34] Dennis Sylvester and Himanshu Kaul. Future performance challenges in

nanometer design. In *Proc. Design Automation Conference*, pages 3–8, NY, USA, 2001. ACM Press.

[35] J. Teifel and R. Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proc. of 10th International Symposium on Asynchronous Circuits and Systems*, pages 17–27, 2004.

[36] A. K. Uht. Uniprocessor performance enhancement through adaptive clock frequency control. In *Proc. of the IEEE Transactions on Computers*, page 132, 2005.

[37] C.H. van Berkel and R.W.J.J. Saeijs. Compilations of communicating processes into delay-insensitive circuits. In *Proc. of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 157–162, 1988.

[38] H. J. M. Veendrick. The behavior of flip-flops used as synchronizers and prediction of their failure rate. In *IEEE Journal of Solid-State circuits, SC-15*, pages 169–176, 1980.

[39] C. G. Wong and A. J. Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *Proc. Design Automation Conference*, pages 508–513, 2003.