Hybrid Synchronous-Asynchronous Tool Flow for Emerging VLSI Design

Filipp Akopyan, Carlos Tadeo Ortega Otero and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca, NY, U.S.A.
faa7@cornell.edu, {cto3,rajit}@csl.cornell.edu

***Abstract*—In the era of high-speed and low-power VLSI circuits, the question of which circuit family is best for a given application has become extremely relevant. From a designer's perspective, technology miniaturization brings increased parameter variation and decreased reliability, which lead to circuit malfunction. To mitigate the risks of undesirable circuit behavior, a designer has to make decisions not only at the micro-architectural scale, but also at the transistor-level scale. Various emerging technologies and non-conventional circuit families may help alleviate reliability problems and provide better performance.**

We have developed automated tools that allow designers to select the circuit family that yields the best results in terms of various design metrics for any application. Using our tools and techniques, the circuit choices can be made in a timely manner without in-depth knowledge of every circuit family under consideration. We demonstrate a tool flow that offers significant reduction in the design cycle time. We provide synchronous and asynchronous circuit libraries for designers to evaluate their circuit architectures and explain how this work can be extended to arbitrary types of circuit families for any given technology node. Finally, we evaluate the novel tools using ITC-99 benchmark suite and present simulation results for various circuit implementations in terms of throughput, power, process corners, and input statistics.

I. Introduction

The main purpose of modern Computer-Aided Design (CAD) tools is to aid designers combat the issues of power management and decreased reliability [1], as well as to decrease design cycle time. Contemporary industrial design flows are well understood, documented, constantly updated and improved by large-scale CAD corporations. However, the limitations of current design flows are also well-known [2], particularly in areas of low-power and high-speed VLSI. The increased complexity of VLSI designs paired with the challenges brought out by feature miniaturization have increased design cycle duration and time to market. One of the primary reasons for this increase is a lack of *fast and accurate* circuit level simulation tools for non standard cell-based circuit families. Presently, if a designer wants to perform gate-level optimizations of a non standard cell-based (or transistor-level optimizations of a standard cell-based) circuit family, 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 many foundries do not reveal the backend structure of the gates in their standard cells to the designers due to confidential intellectual property agreements. In addition, *mixed* high- and low-level simulations are oftentimes complex or even impossible. Thus, designers are forced to perform most of their optimizations only at a high-level circuit scale, where no notions of transistors or even gates exist. This type of decision-making process renders crucial low-level design space optimizations unavailable. Emerging technologies are particularly impacted by the restriction on low-level design optimizations, since existing mapping from high-level description to a limited set of templated cells oftentimes negates the benefits of customized logic. Moreover, if a designer tries to perform low-level optimizations, he is required to have in-depth knowledge of multiple circuit families under consideration. As a result of such complexity, it becomes extremely difficult to predict what circuit family will be better for a given application in a particular environment under a given set of optimization criteria.

An efficient flow would potentially look as follows. A designer specifies his design using a high-level description language, such as Verilog. Then, an intelligent tool automatically synthesizes the description into multiple transistor-level netlists using various types of circuit families, which are then simulated at a transistor-level scale. After analyzing the data obtained from simulations, a designer has enough information to decide which circuit family is best-suited for their design, given their set of requirements.

There are currently dozens of circuit families to consider. Synchronous families include static CMOS, domino logic, differential signaling, etc. Many low-power/high-speed applications may benefit from self-timed (asynchronous) circuit families, which offer tradeoffs in terms of throughput and power consumption in comparison to synchronous circuits [3]. Some potential benefits of asynchronous logic include data-driven switching activity and absence of clock circuitry. However, these advantages come with the overhead of additional handshaking signals and potentially more complex data encoding (for example, dual-rail signals). Asynchronous circuits operate without global clocks and are comprised of many fine-grained hardware processes operating in parallel. Fig. 1 compares the progression in time of synchronous and asynchronous circuit processes. Asynchronous processes operate by communicating 'tokens' using handshake protocols. The data-driven nature of asynchronous circuits allows a circuit to idle without switching activity when there is no work to be done. Another advantage of asynchronous circuits is the capability for correct operation in the presence of continuous and dynamic changes in delays [4]. Sources of such local delay variations may include temperature, supply voltage fluctuations, process variations, noise, and radiation.

In order to allow designers to choose the best circuit family for a design, we have developed a tool flow for automatic synthesis of logic blocks into synchronous and asynchronous logic families. This automatic synthesis enables a systematic comparison between different circuit family implementations.

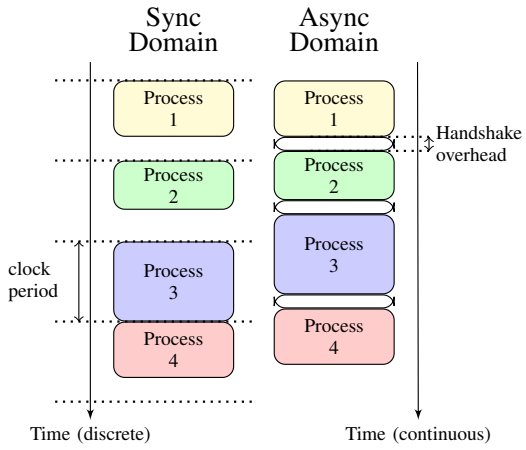

Fig. 1: Synchronous and Asynchronous Domains

After compilation of a given digital logic block into several synchronous and asynchronous implementations, we measure power, performance, and throughput. Using our flow, a designer can evaluate various circuit types and quantitatively determine under which conditions an asynchronous circuit would result in reduced delay or power consumption compared to its synchronous counterpart and vice versa. At this point one can decide which implementation should be used; prior to going through the entire synthesis/layout of all circuit blocks. Our tool flow also provides highly-optimized and pre-compiled cell libraries for different logic families, both synchronous and asynchronous. These libraries eliminate the requirement of thorough knowledge of all circuit families. All our tools are compatible with industrial standard cell libraries – a fact which gives a designer another degree of freedom. One has an option to pick the factory supplied standard cells, if they are sufficient for a given design, or to decide that another circuit family should be used instead.

This paper presents a novel hybrid synchronous-asynchronous tool flow with transistor-level libraries and all the necessary transformations to give a designer the flexibility of performing power/throughput/area optimizations at all levels of the design cycle. We evaluate our tools using the ITC-99 benchmarks, which are commonly used to assess CAD tools.

II. Related Work

Various CAD tools exist for VLSI synthesis using standard cells. Some examples are the industrial synthesis tool offerings from Cadence and Synopsys. Standard tools, such as Synopsys Design Compiler [5], mainly use synchronous static CMOS standard cell synthesis. These tools can take a RTL netlist and synthesize it into synchronous gate-level netlist using a supplied library of standard cells. Since oftentimes foundries do not supply transistor-level descriptions of the standard cells, the flexibility of performing detailed transistor-level optimizations may be taken away from a designer.

Asynchronous circuits can be synthesized using various methods. On one hand, Martin describes a system where CSP-type (Communicating Sequential Processes) programs [6] are decomposed recursively until they can be translated into corresponding transistor-level implementations [7]. On another hand, Tangram [8] and later Balsa [9] use a syntax directed approach to translate CSP to an abstract handshaking circuit, which is then mapped to a standard cell library. While Tangram and Balsa allow a designer to perform prototyping, the resulting circuits tend to be slower and area-inefficient, due to the limited selection of components supplied in the circuit libraries. Farhoodfar *et al.* also use syntax directed translation on hardware processes to synthesize a CSP-like program into a library of templated pipelined buffer cells (PCHB, PCFB) [10]. The resulting circuits compiled by this tool are highly-pipelined and very power hungry.

Another approach to synthesize asynchronous circuits is to leverage existing HDL and synthesis engines to generate asynchronous circuits. One such tool is Pipefitter, which takes an initial specification in Verilog and uses a commercial synthesis engine along with the asynchronous control synthesizer Petrify [11] to generate gate-level netlists. A delay line matched to the latency of the logic in each pipeline stage is added to ensure no race conditions exist. Law [12] presents a similar solution, except with more localized control circuits. Blunno [13] utilizes an approach that involves splitting a system into control and datapath blocks. In order to connect together those blocks, the synthesizable HDL is supplemented with mechanisms to implement asynchronous channels. Ellervee describes the techniques for automatic synthesis of asynchronous circuits for RTL-based design descriptions [14]. Lighthart [15] exploits HDL tools by having a functional library that implements 3-value logic. The resulting RTL undergoes syntactic transformations into CMOS-implementable delay insensitive 2-phase gates. Weaver [16] compiles a synchronous single-rail RTL description into a finely-pipelined QDI asynchronous implementation by replacing synchronous logic gates and registers with their QDI equivalents.

Beerel designed Proteus [17], which is a hybrid approach. The input to Proteus is specified in CSP; the tool goes through a series of syntactic transformations to get an RTL representation, which is then mapped into gates using a single-track full buffer template. Proteus offers significant advantages over other tools as it exploits the expressiveness of CSP and it is complemented with optimizations targeted for asynchronous circuits. The main disadvantage of Proteus is that it requires a complete rewrite of the design into CSP, which makes it unwieldy to perform quick prototyping and comparison against synchronous circuits. Furthermore, Proteus optimizations are limited on large blocks that perform complex algorithmic functions.

To the current knowledge of the authors, the results from previous work have not provided a flexible tool flow, which is evaluated using detailed simulations, analysis and comparison between synthesized asynchronous and synchronous circuits. The extensions of previous tools to utilize *arbitrary* transistor-level circuit families have not been discussed either.

III. Tool Flow Comparison

A. Industrial Tool Flow

A sample industry-standard tool flow for ASIC implementation is shown in Fig. 2. First, a designer creates a high-level Verilog RTL description of the circuits. Second, the RTL is synthesized into a gate-level netlist using a *restricted* set of standard cells with pre-layout timing estimates supplied by the foundry. Third, a designer works with automatic place and route tools to obtain a physical implementation (layout) of the circuit. In practice, the third step is not fully automatic and requires significant manual effort. The generated layout

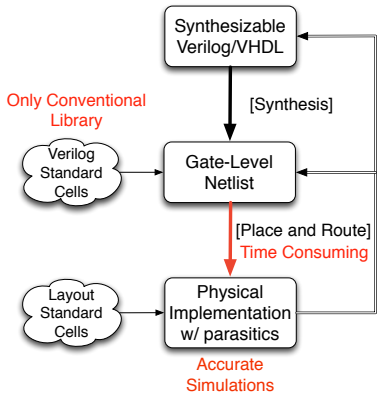

Fig. 2: Industrial Tool Flow

can then be extracted with parasitic effects, with a caveat that oftentimes the back-end contents (transistors) of the standard cells are not revealed. Only after these steps are completed, a designer can perform *accurate* 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 needs to be partially/fully repeated after every modification to the circuits prior to having the ability to perform the next set of accurate simulations. With this flow, it takes designers a *long time* to get to the first (and all subsequent) set of accurate simulations, where many common problems, such as cross coupling, charge sharing, and signal swing issues are revealed.

B. Proposed Tool Flow

In order to reduce the design time and allow engineers to test various types of circuit families for a given implementation, we augment the industrial tool flow, as demonstrated in Fig. 3. Our tool flow contributions are highlighted using green font and surrounded with a dotted box.

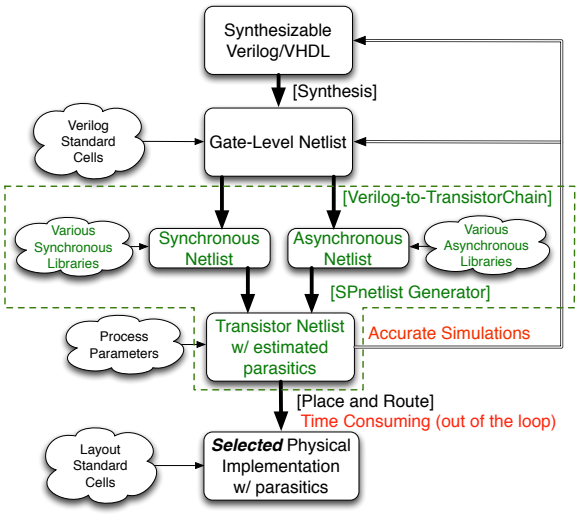

Fig. 3: Proposed Tool Flow

In this flow, a designer creates a high-level Verilog description of the architecture, which is then synthesized into a gate-level netlist using a set of supplied high-level (logic only) standard cells. Any industrial tool, such as Synopsys Design Compiler, may be used for this level of synthesis. At this point we use our custom tool, Verilog-to-TransistorChain (Sec V-A) to generate two netlists: asynchronous and synchronous. Here, instead of using only the "black-box" industrial cells, a designer has a choice of also using highly robust cells from various different circuit family libraries. The circuit family and block granularity is specified in our Verilog-to-TransistorChain tool based on architectural considerations and may be modified at any point. For example, if the selected circuit family for a given block is synchronous, the synthesized netlist is used directly with transistor-level libraries of various synchronous families. However, if the selected family is asynchronous, we perform several netlist transformations (described in later sections) to obtain a logically equivalent asynchronous gate-level netlist. In this case, asynchronous transistor libraries are attached to the generated gate-level netlist. Presently, our tools perform the transformation of synchronous gate-level netlists into Quasi-Delay Insensitive (QDI) [4] asynchronous netlists, but it is simple to perform a similar set of transformations to obtain other types of asynchronous netlists (e.g. bundled-data). A designer *does not* need to have an in-depth understanding of the operation of asynchronous circuits because our tools automatically perform semantic-preserving transformations of the original RTL. Afterwards, we use another custom tool, SPnetlist Generator (Sec V-B) to produce a transistor-level netlist with estimated parasitics for the desired circuit families, which can now be used for accurate simulations. To conduct a realistic simulation, our tools include wiring, fan-out and internal transistor capacitance models (Sec V-B). These models are used to compute parasitics associated with each gate. The advantage of our flow is that, at this *early* point in the design cycle, engineers can perform analog simulations using industrial simulators. These simulation results take into consideration most of the parasitic effects of a given design and aid in making a decision of which circuit family to use for the blocks.

Our proposed tool flow eliminates the iterative place-and-route step for all the preliminary design decisions and measurements. Once the transistor netlist 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 placement related cross-talk, transmission line effects, etc.

C. Proposed Simulator Chain

In our proposed tool flow, a designer has much more flexibility simulating circuits at various pre-layout levels of development, as shown in Fig. 4. As in the industrial flow, the behavioral and RTL Verilog/VHDL code, as well as the gate-level netlist may be simulated with an industrial simulator, such as Synopsys VCS [5]. After we generate the intermediate synchronous and asynchronous netlists, the synchronous netlist may be simulated with the same simulator as before. In the asynchronous scenario, we use our custom digital simulator, PRSIM (Sec V-C). PRSIM may also be used to simulate synchronous netlists, which use custom libraries. We have also developed a simulator interface, Automatic Cosimulation and Environment Generator, to allow the cosimulation of synchronous and asynchronous circuits simultaneously (Sec V-D).

From the synchronous and asynchronous netlists we automatically generate a transistor netlist with estimated parasitics for the block. This netlist is simulated using any accurate

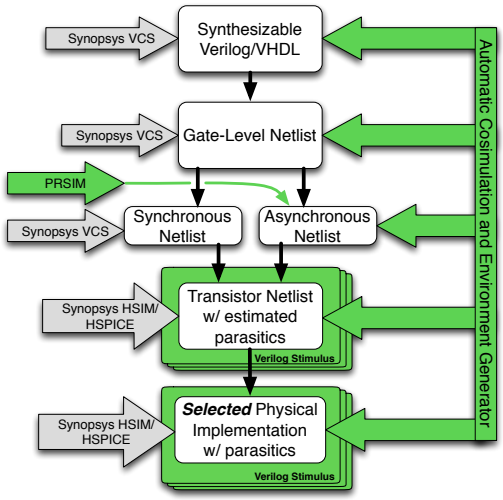

Fig. 4: Sample Proposed Simulator Chain

analog simulator (e.g. HSPICE, Ultrasim, HSIM, Spectre). Majority of the iterative implementation improvement and evaluation tasks are performed at this stage (prior to place and route). After layout is performed, the same analog simulator may be used for the final assessment of the *selected* physical implementation with exact parasitics.

IV. Industrial Tools Used in the Flow

Synchronous Digital Simulator. We use Synopsys VCS for behavioral, RTL, and gate-level simulations. VCS includes an Verilog Procedural Interface, which provides hooks to perform simulations at multiple levels of circuit abstractions.

Transistor-Level Analog Simulator. Synopsys HSPICE is our preferred simulator for accurate circuit simulation. However, we favored Synopsys HSIM for larger circuits [18]. HSIM drastically improves simulation speed because of its hierarchical approach to circuit modeling.

V. Novel Tools Overview

A. Verilog-to-TransistorChain

The Verilog-to-TransistorChain tool converts a Verilog netlist into an equivalent intermediate format gate-level netlist that hierarchically describes pull-up and pull-down transistor chains (called production rules) of each gate used in the design. This tool can be used to generate either a synchronous netlist or an asynchronous netlist. Generation of a synchronous netlist is straightforward. The synthesized Verilog netlist consists of instances of standard cells. Each standard cell can be described using production rules annotated with transistor sizing information. Combining this information with the instances in the Verilog netlist results in a synchronous netlist specified using production rules.

Generation of an asynchronous netlist uses a syntax-directed translation of the synchronous netlist. Each standard cell is replaced with an equivalent asynchronous standard cell, where all input and output wires are replaced by asynchronous handshake channels that carry one-bit data. Combinational logic is replaced by equivalent one-bit pipeline stages. Flip-flops are replaced by one-place token buffers that are initialized with the value of the flip-flop on reset. The main additional step necessary in the asynchronous case is to insert explicit circuits to support fan-out (one-to-many) connections by combining the handshake signals using completion gates. This results in an asynchronous netlist that is structurally equivalent to the synchronous netlist generated by standard logic synthesis. We used this approach to minimize the differences between the synchronous and asynchronous netlists.

The Verilog-to-TransistorChain can work with other asynchronous families (e.g. bundled data) with minimal tool modifications, if required by a designer. Eventually, for more efficient conversion, we would like to perform the synchronous-to-asynchronous transformation using a higher-level behavioral description (where applicable). However, that would require a more complex compiler to perform an optimized and semantic-equivalent translation. Once this high-level asynchronous transformation is implemented, a designer may use other automated methods, such as concurrent pipeline synthesis described by Teifel [19], to perform the circuit synthesis operation. Such high-level transformation is outside of the scope of this work.

B. SPnetlist Generator

This tool syntactically translates every production rule generated by the Verilog-to-TransistorChain into a transistor-level hierarchical SPICE netlist. A single production rule takes the form $G \mapsto S$, where G is a boolean expression called the guard and S is a boolean assignment. Each production rule corresponds to a pull-up or a pull-down transistor switching network, depending on whether the boolean assignment S is for an up-going or a down-going transition. The ordering of transistors is deterministically derived from the production rule in the following manner. Power rails are connected to the source terminal of the transistor generated form the left-most literal in the production rule guard, whereas the output is connected to the drain terminal derived from the right-most literal of the guard. A rule,

```
a & b -> c-
```

is translated into a transistor netlist starting from ground (GND). The '$-$' symbol means that the output describes a down-going transition. The SPICE netlist that corresponds to this production rule is:

```
M0_ GND a #3 GND nfet W=0.2U L=0.045U
M1_ #3 b c GND nfet W=0.1U L=0.045U
```

A configuration file controls multiple parameters such gate input and output capacitances, wiring loads, minimum p- and n- transistor size, source/drain area and perimeter, and spacing between two FETs in the same diffusion stack. From these parameters, the SPnetlist Generator automatically calculates parasitic capacitances and default areas and perimeters for transistor chains. This automatic calculation is crucial to accurately model the behavior of a synthesized circuit.

C. Asynchronous Digital Simulator (PRSIM)

The simulator we use for asynchronous and non-conventional synchronous circuits testing is a custom event-driven digital simulator. The input to PRSIM is an automatically generated (using Verilog-to-TransistorChain) netlist based on production rules. A set of production rules can be viewed as a sequence of events. All events are stored in a queue. When all pre-conditions of an event become true, a timestamp is attached to that event. Once the timestamp of an event coincides with PRSIM's running clock, the event

(production rule) is executed. Timestamps of events can be deterministic or can follow a probability distribution. The probability distributions may be random or long-tailed (i.e most events are scheduled in the near future, while some events - far in the future). Such flexibility allows PRSIM to simulate behaviors of asynchronous and synchronous circuits at random or deterministic timing. Whenever an event is executed, PRSIM performs multiple tests to verify correct circuit behavior, including: 1) Verify that all events are *non-interfering*; an event is non-interfering if it does not result in a short circuit under any allowed input conditions; 2) Verify proper codification on synchronous buses and asynchronous channels; 3) Verify the correctness of expected values of a channel or a bus (optional); 4) Verify that events are *stable*; in asynchronous context, an event is considered stable when all receivers acknowledge each signal transition before the signal changes its value again. We have extended PRSIM's functionality to evaluate energy, power and transient effects of temperature & supply voltage on gate delays, if desired.

D. Automatic Cosimulation and Environment Generator

The Automatic Cosimulation tool allows simultaneous cosimulation of an arbitrary mix of synchronous and asynchronous circuit-families at various levels of abstraction. This tool was developed using two main components: VPI-PRSIM and the custom Environment Generator (which can be also used as a stand-alone tool). The connectivity and interactions between our tools and all the simulators are shown in Fig. 5.

The first component of Automatic Cosimulation, VPI-PRSIM, allows dynamic bindings between the VCS and PRSIM simulations. We have built VPI-PRSIM using the Verilog Procedural Interface (VPI / PLI 2.0) as defined in the Verilog standard [20]. The VPI-PRSIM module provides an API that can be used within Verilog/VHDL to transfer signals and control between a C/C++ program and a Verilog/VHDL simulator. This module allows simulators to exchange timestamps and events through callbacks coordinated by the VPI interface. VPI-PRSIM registers callback functions that are exercised whenever an event occurs. The registered callback functions provided by the VPI interface allow our event-driven simulator PRSIM to process changes in Verilog interface signals. Any time the Verilog/VHDL simulator calls a PRSIM function, PRSIM checks for pending events (signal transitions), updates its event queue and any Verilog signals that are modified by PRSIM. PRSIM synchronizes the simulation timestamp between the two simulators before and after an event is evaluated. The bindings between VCS and HSIM are performed using Synopsys's VCS-HSIM cosimulation interface that also relies on the VPI (PLI 2.0) interface [18].

The second component of our Automatic Cosimulation is the Environment Generator, which automatically creates interfaces between the Verilog/VHDL simulator VCS, the PRSIM asynchronous simulator, and the HSIM transistor-level simulator. It generates the top-level Verilog to interconnect instances defined at various levels of abstraction including Verilog/VHDL, production rules, and transistor-level netlists. The Environment Generator has preloaded communication primitives: boolean signals, buses, and channels. Whenever a connection needs to be made between different levels of abstraction, our tool detects the type of the connection required, and automatically emits the Verilog code that performs the necessary connections. The Environment Generator also allows to back annotate parasitics, specify parameters, interfaces, and initial conditions (ic-s) for PRSIM and HSIM simulators. Furthermore, our tool has an extensive module library that allows designers to automatically send, receive, probe, and check correctness of communication channels and buses. These modules are configurable and can be added on demand using a configuration file.

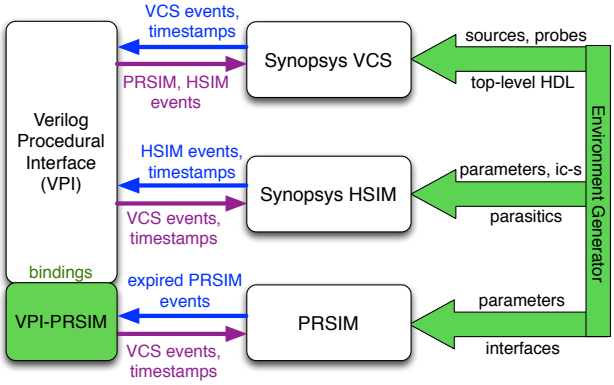

Fig. 5: Automatic Cosimulation Flow Chart

Automatic Cosimulation, with Environment Generator as its key component, allows an engineer to work with hybrid synchronous-asynchronous designs, while implementing the actual circuits using multiple families. A designer can cosimulate high-level behavioral, RTL, production rule, and transistor-level circuit descriptions with a mixture of digital and analog (transistor-level) simulators.

E. Circuit Family Libraries

Our tool flow supports multiple synchronous and asynchronous transistor-level libraries. This feature enables synthesis of circuit descriptions into corresponding transistor-level netlists for various circuit families. Post synthesis (pre-layout) analysis allows designers to select the best circuit family for each part of the architecture depending on the targeted metric, such as power consumption, throughput, latency, etc. For a new technology node it is straightforward to calibrate existing libraries. The parameters that change with a new node are process properties and transistor descriptions: minimum size, parasitic parameters, mobility values, etc. – used for the SPnetlist Generator configuration. As an example, we have created a static CMOS synchronous family and a QDI asynchronous family due to QDI's robustness to delay, temperature, and process variations [21] . We have implemented the QDI cells using PCEHB and HCHB type handshake reshufflings [22].

Layout for the cells in any logic family can be automatically generated by an on-demand std-cell generator, such as custom *cellTK* [23]; or an industrial place and route tool, such as Cadence Encounter. Unlike a traditional ASIC flow, which relies on a predefined and characterized gate-level netlist, *cellTK* generates highly efficient layout cells for an arbitrary transistor-level netlist.

VI. Tool Flow Evaluation

A. Benchmark Considerations

To evaluate our tools, we utilized our two aforementioned libraries: a static highly-robust synchronous library (industry-

based) and a QDI PCEHB-based asynchronous library. The circuits are implemented in a 45 nm technology node [24].

We have obtained ITC-99 benchmarks [25,26] and compiled them into synchronous and asynchronous netlists using our tool flow. We selected the ITC-99 benchmarks because of their variety in functionality, structure, and complexity as shown in Table I. The benchmarks range from simple sets of gates to designs with around 600 gates and dozens of flip-flops. Some benchmarks represent control processes, while others embody complex arithmetic and logical data paths. The behavioral and RTL design of the ITC-99 benchmarks is originally created in VHDL, which is also supported by our tool flow.

TABLE I: ITC-99 Benchmarks and Functionalities

Name	Description	Gates	Flip-Flops	I/O ports	VHDL lines
b01	Serial flow comparator	45	5	4/2	110
b02	BCD number recognizer	25	4	3/1	70
b03	Resource arbiter	150	30	6/4	141
b04	Min-Max search	480	66	13.8	80
b05	Memory	608	34	3/36	319
b06	Interrupt handler	66	9	4/6	128
b07	Count points on a straight line	382	51	3/8	92
b08	Numeric series	168	21	11/4	89
b09	Serializer/Deserializer	131	28	3/1	103
b10	Voting system	172	17	13/16	167

B. Throughput Comparison

To measure the throughput of the asynchronously implemented benchmarks, we run transistor-level simulations of the circuit with random input patterns for a sufficient amount of time to reach steady state and then capture the average throughput. All simulations are initially performed using the typical-typical (TT) transistor corner models. For the synchronous implementations, we initially run the clock at a low frequency for a long period of time and record the trace of output values, using random inputs. We then gradually increase the clock frequency and compare the obtained values with the trace recorded initially. Whenever we find a mismatch in the output values or internal state, we postulate 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 regarded as the maximum circuit frequency under the given conditions. All the original synchronously-implemented ITC-99 benchmarks have *ideal* clocks, i.e. they do not account for clock distribution and clock uncertainties such as signal jitter and skew. As a result, to perform a fair comparison, we padded the synchronous operating frequency with a 20 % safety margin, which is consistent with studies found in the literature [27].

The average absolute throughput in MHz for both synchronous and asynchronous implementations is shown in Fig. 6, in the columns labeled *Sync-Nominal* and *Async-Nominal*. The synchronous implementations perform better for less complex benchmarks [b01, b06], while more complex benchmarks gain throughput using asynchronous realizations [b05, b07].

C. 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 extreme process corners: slow-slow (SS) and fast-fast (FF). Fig. 6 also demonstrates the effect that process corners have on the behavior of synchronous and asynchronous circuits respectively.

As expected, the throughputs of both circuit families increase when both NFET and PFET devices get faster. On average, the designs running in the FF process corner deliver an average of 27% more throughput than in the TT corner. On the other hand, when transistors are running in the SS process corner, the benchmark performance degrades by an average of 25% in this process technology. Synchronous circuits tend be slightly less sensitive to process variation in terms of performance compared to their asynchronous counterparts, as demonstrated in Fig. 6. However, what is not captured by these results is that synchronous implementations must account for the worst possible variation, while asynchronous implementations *automatically* adjust to the wide range of process variations. The trends for the SS and FF corners exactly resemble the nominal scenario, considering the relative speed variation. 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 variations considered in this study are 3σ, represented by the process corners.

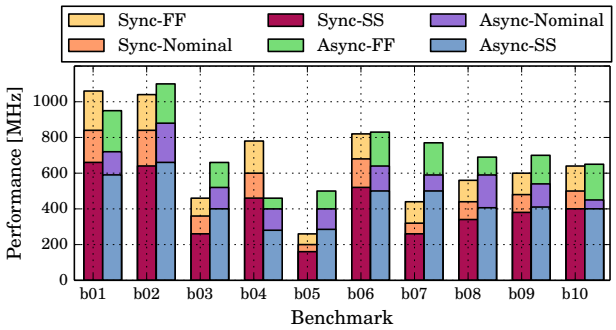

Fig. 6: Performance and Process Variations Analysis

D. Power Analysis

In our power experiments, we want to find the effective input signal duty cycle that results in equal power consumption of the synchronous and asynchronous implementations, while still maintaining the ability of processing bursty inputs at the maximum throughput. For that, we fix the throughput of both synchronous and asynchronous circuits (per benchmark) to the highest frequency that both implementations support nominally, i.e. lower value of the {Sync-Nominal, Async-Nominal} pair in Fig. 6. We then vary the input signal duty cycle by sending bursts of high-throughput inputs, followed by periods of input inactivity. Our analysis assumes that the power consumption is dominated by the dynamic power. This proves to be correct in low leakage technologies, where similar power consumption is observed for the presented synchronous and asynchronous implementations.

At the scale of benchmarks in our analysis (small and medium size circuits), the overheads of clock-gating control complexity offset the benefits from clock-gating itself. We, thus, compare the asynchronous implementations to non clock-gated synchronous implementations. Also, due to the fact that clock tree networks were not physically present in the original benchmarks, we have omitted the clock tree load and switching energy for all synchronous benchmarks.

Mathematically, the dynamic power estimated by Eq. 1, is used to derive Eq. 2 (for asynchronous circuits) and Eq. 3 (for synchronous circuits). In the following calculations f_{sw} is the circuit's switching frequency, which is estimated with the frequency of circuit operation in each scenario. In the asynchronous case, Eq. 2, α_{async} is the activity factor of the circuit, C_{async} represents the total load capacitance in the asynchronous implementation and f_{avg} is the input frequency averaged across all bursts and inactivity periods. f_{avg} is also representative of the average duty cycle of the inputs. The activity factor in a QDI asynchronous design is maximally equal to 1, since majority of the nodes in an asynchronous circuit switch twice per handshake by going to active phase and neutral phase. In the synchronous calculation, Eq. 3, there are two components contributing to the circuit's dynamic power: the combinational logic term and the clock term. Here, C_{cl} is the lumped load capacitance of the logic, and C_{clk} is the clock network capacitance. α also has two components: α_{cl}, which is the activity factor of the logic and, for consistency, α_{clk} - the activity factor of the clock. f_{clk} denotes the high-throughput signal switching within input bursts and, in the synchronous case, represents the clock frequency required to support this high throughput. f_{cl}, the switching frequency of combinational logic is equal to $\alpha_{cl} f_{avg}$, since the combinational logic needs to switch only when inputs change (which in the case of bursty inputs does not happen on every clock cycle). α_{cl} is maximally equal to 0.5 for combinational logic according to statistics and α_{clk} is equal to 1, since a clock has two edges per cycle.

$$P = C_{total} V dd^2 f_{sw} \tag{1}$$

$$\begin{aligned} P_{async} &\approx \alpha_{async} C_{async} V dd^2 f_{avg} \\ &= C_{async} V dd^2 f_{avg} \end{aligned} \tag{2}$$

$$\begin{aligned} P_{sync} &\approx C_{cl} f_{cl} V dd^2 + C_{clk} f_{clk} V dd^2 \\ &= (\alpha_{cl} C_{cl} f_{avg} + \alpha_{clk} C_{clk} f_{clk}) V dd^2 \\ &= (0.5 C_{cl} f_{avg} + C_{clk} f_{clk}) V dd^2 \end{aligned} \tag{3}$$

We can get a closed form solution for approximate value of f^*_{avg} (power break-even average input frequency for synchronous and asynchronous implementations) by equating Eq. 2 and Eq. 3 and solving for f_{avg}. The obtained result is given by Eq. 4 and shows that f^*_{avg} is directly proportional to the product of 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 combinational logic capacitance C_{cl}.

$$f^*_{avg} = \frac{C_{clk} f_{clk}}{C_{async} - 0.5 C_{cl}} \tag{4}$$

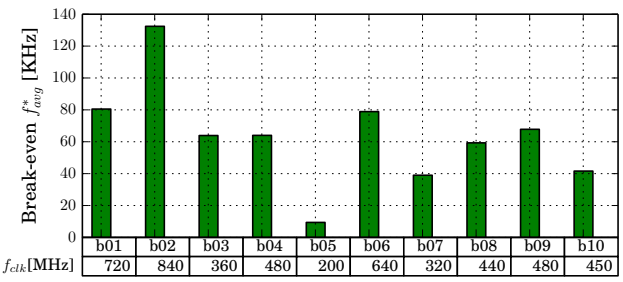

Fig. 7: Power Break-Even Average Input Frequency

Fig. 7 shows the power break-even average input burst frequency f^*_{avg} for each benchmark. Specifically, at f^*_{avg} the power consumption of the asynchronous implementation is equal its synchronous counterpart. f_{clk} in this figure is shown in MHz. In the analysis of Fig. 7, all the capacitances were extracted from the actual circuits (while omitting the clock distribution network). One can see that average break-even frequency occurs in the 10-s to 100-s of KHz range for different benchmarks (while still supporting 100-s of MHz maximum input throughput). For bursty, yet high-throughput inputs within the busts, asynchronous circuits provide a better tradeoff in terms of power consumption below this f^*_{avg}. Above f^*_{avg} synchronous circuits are more power efficient for our presented set of gate-level netlist transformations. To improve the power efficiency of asynchronous circuits in our tool flow we are looking into implementing several additional techniques, as outlined in Sec VI-F.

E. Design Space Investigation

Our tools demonstrate the tradeoffs of implementations in the synchronous and asynchronous design spaces. An example study is shown in Fig. 8, which presents a 3-dimensional view of the average input frequency vs. maximum supported throughput vs. calculated dynamic power consumption for benchmark b01. Fig. 8(a,b) demonstrates power consumption for the asynchronous and synchronous implementations respectively. Similar to previous experiments, f_{avg} denotes the input frequency averaged across bursts and inactivity periods on y-axis, f_{clk} denotes the maximum signal throughput within the bursts on x-axis, and the color gradient represents the power consumption. Fig. 8(c) shows the difference in dynamic power, P_{async} and P_{sync}, for the two implementations. One observes that synchronous realization has less power consumption in the presence of constant high input data rates represented by f_{avg}. In contrast, the asynchronous implementation results in a more efficient power consumption when maximum throughput is required (f_{clk}), but the average input frequency is low (f_{avg}). The graphs for the other benchmarks follow a similar trend. The only difference in the other benchmarks is the absolute value on the break-even f^*_{avg}, as shown in Fig. 7.

F. Analysis of Simulation Results

The obtained results agree with the original hypothesis that we have made while developing these synchronous-to-asynchronous transformations. In many cases, the obtained asynchronous implementations were able to achieve higher performance than their synchronous counterparts. Additionally, our initial prognosis, which turned out to be true, stated that due to the nature of the transformations (gate-level netlist conversion), the number of transistors in the asynchronous implementation would on average be higher than that of the corresponding synchronous implementation (the resulting area overhead of asynchronous circuits was at least twice the area of the synchronous circuits). Similarly, in the QDI 4-phase handshake circuits the signal activity factor is higher, since the circuit on every data token goes through the active phase and returns to the neutral phase. These facts lead to a higher power consumption of the asynchronous implementations in scenarios, where the inputs arrive at high frequency with no inactive periods. However, the asynchronous implementations automatically produced in this tool flow have proven to be advantageous in designs, where there are bursts of high-frequency

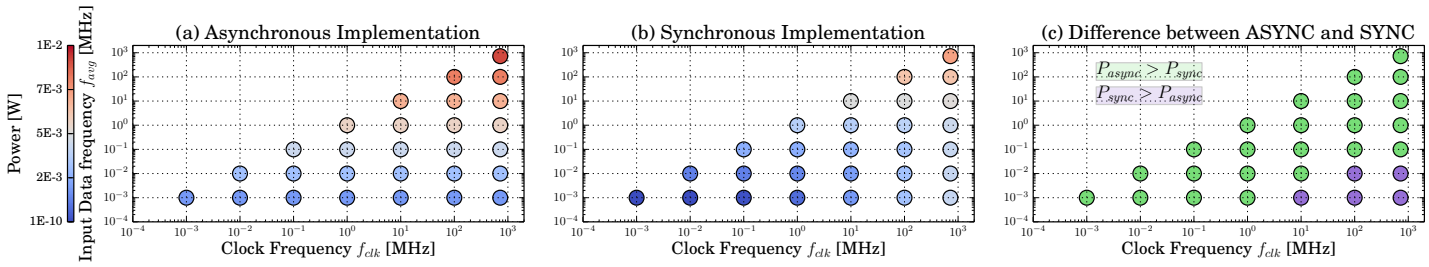

Fig. 8: b01 benchmark Power Consumption (in Watts) for sweeps of f_{avg} and f_{clk} frequencies

inputs followed by long quiescent periods. Such applications include speech processing, on-chip networks, neuromorphic circuits, etc. In these designs, the circuit irealizations must still be able to support maximum throughput, which means that the clocks would have to switch at the maximum input rate if implemented in a synchronous manner.

We are planning to implement several additional techniques to enhance out tool flow. Specifically, we are looking into optimizing the gate-level transformations and pipelining based on designer's metrics (area, throughput, power). To perform this effectively we would extend our tool flow and add libraries for other synchronous and asynchronous circuit families. Beyond that, as mentioned previously, we want to implement high-level transformations to allow implementation-specific optimizations earlier in the design cycle.

VII. Conclusion

In this paper, we proposed a novel way of designing complex VLSI circuits. This approach is especially useful in large-scale projects where accurate measurements and decisions early in the design cycle can drive many high-level architectural decisions. Our tool flow allows designers to select the ideal circuit family for each digital block in the design based on various metrics, without necessitating thorough expertise in all logic families. The tool flow may be extended with additional synchronous and asynchronous circuit family libraries without the necessity to modify the underlying tools.

As part of the novel flow, we also presented a method for integrated simulation framework of both synchronous and asynchronous circuits. This framework allows cosimulation using different circuit technologies and different levels of abstraction. To evaluate the framework, we provided quantitative results comparing multiple benchmark implementations using two logic families: QDI asynchronous and static synchronous. The presented gate-level pipelining approach to compile asynchronous circuits increases the throughput compared to the synchronous approach, while also increasing power consumption and area in many cases. However, for applications where the circuits are mostly idle and have bursts of high frequency activity, the power consumption of the asynchronous implementations is lower. In the future, among other enhancements, we would like to focus on performing not only gate-level syntactic transformations, but also higher-level, semantic-preserving circuit family-specific architectural optimizations and transformations.

References

[1] D. Sylvester and H. Kaul, "Future performance challenges in nanometer design," in *Proc. Design Automation Conference*, ACM Press, 2001.

[2] R. Bryant and et al., "Limitations and challenges of computer-aided design technology for CMOS VLSI," in *Proc. of the IEEE*, 2002.

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

[4] A. J. Martin, "The Limitations to Delay-Insensitivity in Asynchronous Circuits," in *ARVLSI*, 1990.

[5] "Synopsys tool manuals," in *https://solvnet.synopsys.com/*, 2010.

[6] C. van Berkel and R. Saeijs, "Compilations of communicating processes into delay-insensitive circuits," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1988.

[7] S. Burns and A. Martin, "Syntax-directed translation of concurrent programs into self-timed circuits," tech. rep., DTIC Document, 1988.

[8] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *EDAC*, IEEE, 1991.

[9] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, no. 1, 2002.

[10] H. P. K. Saleh, M. Naderi, M. H. Shafiabadi, H. Kalantari, and A. Farhoodfar, "Synthesis Tool for Asynchronous Circuits Based on PCFB and PCHB," in *Proc. Computer Society of Iran Computer Conference*, 2004.

[11] J.Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers,"

[12] C. Law, B. Gwee, and J. Chang, "Modeling and synthesis of asynchronous pipelines," *IEEE Trans. Very Large Scale Integr. Syst.*, 2011.

[13] I. Blunno and L. Lavagno

[14] J. Oberg, J. Plosila, and P. Ellervee, "Automatic synthesis of asynchronous circuits from synchronous RTL descriptions," in *NORCHIP Conference*, 2005.

[15] M. Lighthart, K. Frant, R. Smith, A. Taubin, and A. Kondratyev

[16] A. Smirnov, A. Taubin, M. Su, and M. Karpovsky, "An automated fine-grain pipelining using domino style asynchronous library," in *ACSD*, 2005.

[17] 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*, 2004.

[18] Synopsys, Inc., "HSIM^{PLUS} © Reference Manual," 2011.

[19] J. Teifel and R. Manohar, "Static tokens: Using dataflow to automate concurrent pipeline synthesis," in *International Symposium on Asynchronous Circuits and Systems*, 2004.

[20] "IEEE standard for Verilog hardware description language," *IEEE Std 1364-2005*, 2006.

[21] A. M. Lines, "Pipelined asynchronous circuits," tech. rep., Caltech, 1998.

[22] 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*, 2009.

[23] R. Karmazin, C. O. Otero, and R. Manohar, "cellTK: Automated layout for asynchronous circuits with nonstandard cells," in *International Symposium on Asynchronous Circuits and Systems*, 2013.

[24] "Nangate 45nm open cell library," in *http://www.nangate.com/*, 2009.

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

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

[27] A. K. Uht, "Uniprocessor performance enhancement through adaptive clock frequency control," in *IEEE Transactions on Computers*, 2005.