Toward a digital flow for asynchronous VLSI systems

Samira Ataei†, Jiayuan He*, Wenmian Hua†, Yi-Shan Lu*, Sepideh Maleki*, Yihang Yang†, Keshav Pingali*, and Rajit Manohar†

*University of Texas at Austin
{yishanlu,hejy,smaleki,pingali}@cs.utexas.edu

†Yale University
{samira.ataei,wenmian.hua,yihang.yang,rajit.manohar}@yale.edu

***Abstract*—We are developing an open-source EDA flow for asynchronous logic. Key parts of the flow are implemented using the Galois system for parallelization to reduce run-time requirements. We report on the current state of this flow, and some of the open issues that we are exploring in order to improve the overall quality of results.**

I. Introduction

Asynchronous digital circuits provide a method for implementing digital computation without the need for a global clock signal to synchronize all operations. The absence of a global clock is a significant enough change that conventional synchronous design automation tools are insufficient to design correct asynchronous circuits without major manual effort.

To address this challenge, we have been building an open-source design automation flow for asynchronous circuits. The flow we are creating leverages over three decades of experience in the asynchronous design community in manual design and fabrication of chips that exhibit state-of-the-art power/performance. The approach starts with the circuit specified using a programming language called "Communicating Hardware Processes (CHP)" which is a combination of Hoare's classic CSP [3] and Dijkstra's guarded command language [1]. The language is a sequential programming notation augmented with communication channels that support send, receive, and probe operations. Key design decisions are made by re-writing the CHP program into a massively parallel collection of "small" CHP programs. This decomposition of the original program is where all micro-architectural decisions (e.g. pipelining, resource usage, computation/storage trade-offs, etc.) are made. Many systematic techniques have been developed for this purpose, including process decomposition and projection [9, 10].

The final CHP program is elaborated into operations on Boolean-valued variables, and finally these programs are synthesized into a collection of CMOS gates. After this step, electrical optimizations such as gate sizing and buffer insertion can be applied. Each gate has to be converted into a VLSI layout (a "cell"), and then these cells have to be placed and routed. In support of the flow, we also plan to develop formal equivalence checking tools across different levels of abstraction (e.g., CHP versus gates) using a combination of traditional model checking and *inverse synthesis*, which attempts to "undo" steps in the design flow [7, 8].

In this paper, we describe the progress we have made over the past year in developing design automation support for asynchronous circuits. We focus on the "gates to GDS2" part of the flow, where a gate-level netlist has to be translated into physical geometry prior to tape-out. The modular nature of the asynchronous design flow makes it amenable to parallelization. Module-level parallelism is easy to achieve, and the higher complexity of some of the design automation steps (relative to synchronous design) requires parallelism to speed up the design process. We are leveraging the Galois framework as a way to simplify the development of highly parallel EDA tools.

II. EDA for Asynchronous VLSI Systems

Just like there are many synchronous logic families that each have their benefits and drawbacks, there are a large number of different asynchronous logic families as well. While they have significant differences and provide differing points in the power/performance/area design, space, they have similarities and differences when compared to conventional synchronous logic. In what follows, we assume the target implementation technology is CMOS.

Both synchronous and asynchronous circuits are translated to a collection of CMOS-implementable gates. These gates are implemented with transistors used as switching networks. However, asynchronous circuits use more general gates than those found in standard commercial cell libraries. As an example the C-element, a commonly used asynchronous state-holding gate, is absent from cell libraries.

Calculating gate delays and translating gates and interconnections into a physical VLSI implementation is the same in both synchronous and asynchronous circuits, since they both use the same (CMOS) implementation technology with the same manufacturing rules. Electrical simulations using SPICE simulation is also the same, since the circuit simulation uses the same electronic components/devices in both asynchronous and synchronous logic.

However, there are two major differences in performance computation. Synchronous logic implemented with flip-flops and combinational logic concerns itself with the maximum

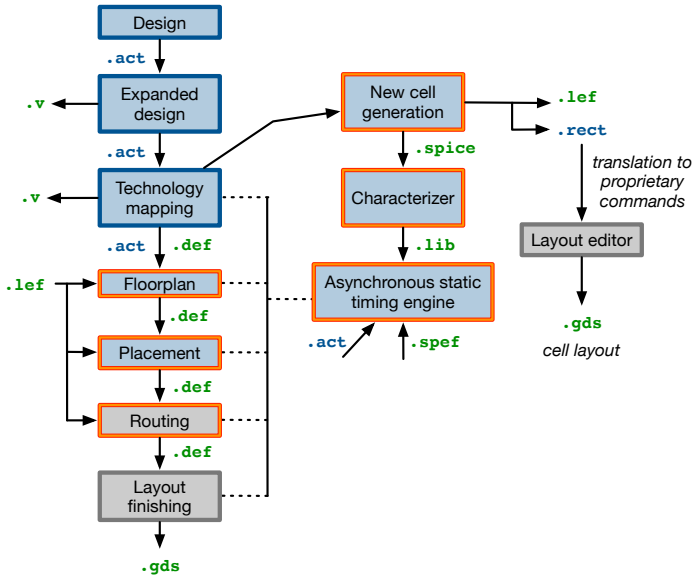

Fig. 1: Overview of the digital design flow for asynchronous circuits that is under development. Gray boxes denote existing tools that we leverage. Orange boxes are the ones where we implement parallel algorithms to improve run-time.

and minimum delay through the combinational logic, and the setup and hold times of state-holding flip-flops. Instead, asynchronous circuits' performance is governed by the delays of cycles of gates. Hence, timing analysis is quite different in the asynchronous domain. Also, different asynchronous circuit families have different timing requirements for correct operation. Therefore, timing analysis tools for asynchronous logic differ significantly from those needed for synchronous logic. This impacts gate sizing, buffer insertion, and placement and routing—in other words, the physical implementation flow.

The flow we are creating includes a design language called ACT (for asynchronous circuit toolkit). This is a hierarchical design language that includes communication channels as first-class objects. The language supports representing circuits at multiple levels of abstraction, including CHP, gate-level, transistor-level descriptions, and abstract layout/geometry. By using an integrated language, we preserve the relationships between different levels of abstraction in the design throughout the design flow. Design tools can be viewed as transformations in the ACT framework. For example, logic synthesis elaborates a CHP-level description of a module into a gate-level description of the *same module* without changing its interface. This language is the result of an evolution over almost three decades of research in asynchronous design grounded in the implementation of over a dozen asynchronous VLSI chips ranging in complexity from 0.5M transistors to 5.4B transistors, and in technologies ranging from 0.6μm CMOS to 28nm CMOS. A summary of the flow we are developing is shown in Figure 1.

A. Design elaboration

The ACT language permits parameterized and templated designs. Hence, the first step in the automation flow is to elaborate the design and create an instance of the design corresponding to the specific parameters selected for implementation by the designer. This transformation is referred to as *expansion*, and the expanded design is also represented in the ACT framework. After the expansion of the design is complete, it is possible to export the design into a Verilog netlist format for ease of integration with commercial tools or other third-party open-source tools.

B. Technology mapping and gates

The expanded design contains a hierarchical description of the asynchronous circuit. As we have alluded to earlier, asynchronous circuits are implemented using a wide variety of gates. To support this using a digital flow, we have implemented a gate isomorphism pass.

Given a collection of existing/known gates/cells, the gate isomorphism pass matches the gates used by the user design against these known cells. The known cells are specified using the ACT language and are encapsulated in a special "`cells`" namespace in ACT. If there is a gate in the user design that cannot be found, then a new cell is added to the `cells` namespace. This new cell is converted into layout using a cell generation approach we have previously developed [5], and then characterized using the `Xyce` open-source circuit simulator. The characterization phase results in the creation of a new `.lib` file segment for timing analysis. In addition, `.lef` files are generated for placement and routing. The geometry for the cell is created using a very simple `.rect` file format, which consists of a simple list of rectangles, layer names, and (possibly empty) labels.

Since many commercial layout tools use proprietary commands/formats, we adopted this simple approach so that it would be easy to translate the geometry generated by our tools into scripts/commands to re-create the same geometry within any commercial tool. As an example, our Perl script that translates the `.rect` file into a script that can be used by the `magic` VLSI layout editor is twenty-three lines long.

The cell layout generator is parameterized by layout design rules that are sufficient to generate correct cell geometry in technologies upto 28nm. We are currently examining the necessary extensions for 14nm and below. Since the generator is internally paramterized, we can either generate fixed height cells (akin to commercial cell libraries), or cells that can have variable height and width.

Note that cell layout generation and characterization is an embarassingly parallel problem, and since we use open-source tools for all of these steps we are not limited by licenses in order to exploit the available parallelism.

C. Asynchronous static timing analysis

Once we have characterized gates and a design that is mapped to a collection of pre-characterized gates, we can

perform asynchronous static timing analysis. Our timing analysis engine adopts the mathematical framework developed by [4], as it can handle a wider class of asynchronous circuit topologies. The approach is currently limited to asynchronous circuits with max-causality gates, and calculates conservative approximations in all other cases. We are currently extending this to handle min-causality, as well as more complex cases.

Our timing analysis approach analyzes cycles of gates and computes the cycle period of the asynchronous circuit, as well as the timing slack for each branch in the circuit. Compared to synchronous timing analysis, the asynchronous timing analysis engine takes roughly two orders of magnitude more time for designs of comparable size (in terms of gate count and net count) for quasi delay-insensitive circuits.

To address the performance issue, we use the Galois framework (summarized in the next section) to parallelize this algorithm so as to keep the run-time manageable. In our preliminary experiments, we obtain a speedup that is roughly a factor of $5\times$ to $10\times$ for large designs by using the Galois framework. We are currently examining other approaches to improve the speedup further, as well as improving the accuracy of asynchronous timing analysis.

D. Floorplanning and placement

Floorplanning is automated for larger designs by using a hierarchical min-cut approach. Both layout space and the design are simultaneously partitioned while keeping the ratio of cell area to available space constant.

Each floorplanned region is then placed and routed using a variation of existing analytical placement engines. Placement proceeds with global placement to minimize half-perimeter wirelength, followed by detailed placement which also attempts to minimize overlaps, followed by legalization that eliminates all overlaps. We have extended existing approaches to legalization to handle cells with varying heights. Timing slack computed from timing analysis is used to improve the quality of placement, and adapt it to the requirements for asynchronous logic. This is currently a work in progress, as we evaluate the quality of results of our approach for a range of asynchronous circuit families and CMOS technologies.

E. Routing

We take the `.def` file generated from placement and the `.lef` files from cell generation and use these as inputs to a detailed router. Currently we use the open-source `qrouter` to route the placed design, but our approach is agnostic to the choice of detailed router.

We have parallelized the FastRoute global router [14] using the Galois framework [2]. We plan to integrate this global router into the layout automation framework.

Our current plan is to modify the detailed router to make it timing driven, using feedback from the asynchronous timing analysis engine that has been developed.

III. The Galois framework

Since circuits can be viewed abstractly as graphs and hypergraphs, a system for supporting the design and implementation of a parallel EDA tool-chain can be implemented using a framework that directly supports parallelism for graph-based algorithms. The Galois system is one such framework, and what is unique about it is that it exposes parallelism using the operator formulation of algorithms [12].

The Galois system implements a data-centric programming model (see details in [12]). Application programmers write programs in sequential C++, using certain programming patterns to highlight opportunities for exploiting amorphous data-parallelism. The Galois system provides a library of concurrent data structures, such as parallel graph and work-list implementations, and a runtime system. The data structures and runtime system ensure that each activity appears to execute atomically. In this way, the Galois system encapsulates parallelization details and realizes performance scalability at the same time.

The Galois system has been used to implement parallel programs for many problem domains including finite-element simulations, n-body methods, graph analytics, intrusion detection in networks [6], FPGA routing [11], and AIG rewriting [13].

For the effort we are describing, we have successfully used the Galois framework for parallel implementations of:

- floorplanning, through a parallel implementation of hierarchical min-cut partitioning;
- global routing, by exploting both net-level parallelism, as well as parallel maze routing; and
- timing analysis, by implementing parallel timing propagation.

IV. Conclusion

We presented a plan for implementing a parallelized physical design flow for asynchronous circuits. The flow supports modular design and parallelization naturally. We believe that this tool chain will promote chip design with clean abstractions and fast turn-around times.

References

[1] Edsgar W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, pages 453–457, 1975.

[2] Jiayuan He, Martin Burtscher, Rajit Manohar, and Keshav Pingali. Sproute: A scalable parallel negotiation-based global router. In *International Conference on Computer-Aided Design*, November 2019.

[3] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[4] Wenmian Hua and Rajit Manohar. Exact timing analysis for asynchronous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):203–216, 2018.

[5] Robert Karmazin, Carlos Tadeo Ortega Otero, and Rajit Manohar. celltk: Automated layout for asynchronous circuits with nonstandard cells. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 58–66. IEEE, 2013.

[6] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Commun. ACM*, 59(5):78–87, April 2016.

[7] Stephen James Longfield and Rajit Manohar. Inverting martin synthesis for verification. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 150–157. IEEE, 2013.

[8] Stephen Longfield Jr and Rajit Manohar. Removing concurrency for rapid functional verification. In *Proc. 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 332–339. IEEE Press, 2014.

[9] Rajit Manohar, Tak-Kwan Lee, and Alain J Martin. Projection: A synthesis technique for concurrent systems. In *IEEE International Symposium on Asynchronous Circuits and Systems*, page 125. IEEE, 1999.

[10] Alain J Martin. Synthesis of asynchronous vlsi circuits. Technical Report CS-TR-93-28, California Institute of Technology, 1993.

[11] Yehdhih Ould Mohammed Moctar and Phillip Brisk. Parallel fpga routing based on the operator formulation. In *DAC '14: Design Automation Conference*, 2014.

[12] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[13] Vinicius Possani, Yi-Shan Lu, Alan Mishchenko, Keshav Pingali, Renato Ribas, and Andre Reis. Unlocking fine-grain parallelism for aig rewriting. In *ICCAD '18: International Conference on Computer Aided Design*, 2018.

[14] Yue Xu, Yanheng Zhang, and Chris C. N. Chu. Fastroute 4.0: Global router with efficient via minimization. *2009 Asia and South Pacific Design Automation Conference*, pages 576–581, 2009.