interact: An Interactive Design Environment for Asynchronous Logic

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

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

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

***Abstract*—We are developing an open-source EDA flow for asynchronous logic. We present the current state of the flow, where all the key components have been integrated into a single framework including the timer, partitioner, placer, power detailed router, and global router. We describe enhancements to the flow in terms of the class of circuits that can be handled, and extensions to support third-party libraries and flows.**

I. Introduction

Asynchronous logic is an approach to digital design without using clock signals to orchestrate the computation. This change presents major challenges to mainstream design automation tools that are designed with the clocking assumption implicit.

We have been developing an open-source design automation flow for asynchronous circuits. The flow includes an open-source hardware description language dubbed "ACT" (for Asynchronous Circuit Tools/Toolkit), which permits the specification of asynchronous circuits at multiple levels of abstraction ranging from behavioral descriptions at one end and transistor-level descriptions at the other. The ACT language supports different asynchronous circuit families and timing constraints in a single framework.

We previously reported on a number of tools developed to support asynchronous design, with our philosophy being that we would only like to develop tools that are specific to the asynchronous design methodology while re-using existing synchronous tools where possible. In this work, we report on a design environment that we have developed over the past year that integrates all our tools into a single unified framework. We call this tool `interact`, as it provides an interactive command-line interface to the ACT framework. This tool integrates the following pieces of our flow that are asynchronous design specific: design management/elaboration, cell mapping, timing analysis, partitioning, global and detailed placement, power detailed routing, and global routing.

`interact` also permits the exchange of information between different tools, and in particular, between timing analysis and other physical design steps. We describe how we achieved this in an academic environment, where different student research projects tend to be "loosely coupled."

II. Core data structures

The design information is specified in the ACT open-source hardware description language. The language is available at `http://github.com/asyncvlsi/act/`.

ACT maintains design information in a hierarchical fashion, and provides application programming interfaces (APIs) to access circuits, components, wires, and cells. We use the hierarchical ACT data structures as the "master copy" of a user's design information.

As our previous place and route tools used LEF/DEF as their input/output format, we implemented a physical database layer called *PhyDB* that represents the "master copy" of physical geometry information. *PhyDB* can be viewed as an in-memory copy of LEF/DEF, and is similar to *OpenDB* [9] but specialized for ACT and supports a more recent LEF/DEF version needed for advanced technology nodes. PhyDB provides three types of APIs:

- LEF/DEF file APIs, to populate the database from files and export it to files.
- Access and update APIs, for physical design tools to pull input from the database and push output to it without having to use files.
- Timing APIs, for connecting the timer to other physical design tools.

The LEF/DEF file APIs are implemented using the Si2 LEF/DEF parser. The access and update APIs are used by physical design tools including placers and routers. For example, the ACT flow is able to add cell libraries, components, and netlist. Placers can change the cell locations and router can add wires and vias to nets. Timing APIs report violated timing constraints and their corresponding slacks computed from the timer for timing-driven placement and routing.

Information flow between ACT and *PhyDB* is orchestrated by the `interact` tool. After design elaboration, the physical design problem is constructed from the ACT design and pushed to *PhyDB*.

A. Individual components

The key existing tools that we have integrated into `interact` are: BiPart, a parallel partitioner [6]; Dali, a gridded cell global and detailed placer [10]; SPRoute, a parallel

global router [3]; PWRoute [4], a power detailed router; and Cyclone, a static timing analysis engine [5]. To do so, each tool was converted into a library with API calls that implemented the functionality of the original executable. These libraries are all linked with the `interact` binary. The physical design tools were already using the open-source LEF/DEF parser from Si2; hence, it was a straightforward change to modify them to access the information from *PhyDB*.

B. Extensibility

The core ACT library was developed with extensibility in mind. ACT transforms a design internally by executing a number of "passes," similar to how a software compiler executes passes to re-write programs. Examples of such passes include gate sizing, netlist generation, cell mapping, and transistor mapping. In addition, we added the capability to load in passes at run-time via a shared object library, similar to the mechanism that exists in the LLVM software compiler infrastructure [1]. To ensure this mechanism is sufficiently expressive, our timing graph construction pass and LEF/DEF generation pass were written as external libraries that are dynamically loaded into `interact` via run-time commands.

III. Netlist Adaptor

Timing-driven physical design involves the interaction among timing analysis, placement, routing, gate sizing, and buffering. How to represent a netlist to fit all tools' needs is challenging, as each tool tracks and manipulates different aspects of a given gate-level netlist. For example, a timer computes event timing using pin connectivity, liberty files, and RC trees for nets; a placer positions cell instances while considering wire lengths of nets; and a router connects pins by realizing nets through positioning metal segments and vias.

One can force all tools to work on a unified netlist representation. However, doing so couples all tools through the netlist representation, so it will be more difficult to extend and maintain the tools in the future. Moreover, this hurts the performance of EDA tools by introducing bad locality of data fields in a netlist, and by consuming more memory space for fields not needed.

All gate-level tools require a netlist representation to support the following functionalities:

- Get the full name for a given object, i.e., a pin, a net, or a cell instance. This is for file output.
- Get an object given its full name. This is for file input.
- Given a pin, get the net/cell instance that the pin belongs to. This is to check pin connectivity through nets.
- Given a pin, get its name in its belonging cell instance. This is to check pin connectivity in a cell instance.
- Given a cell instance, get its cell type name. This is to check pin connectivity in a cell instance.
- Test if two objects are equal. This can be used to handle object aliasing.
- Compare two objects. This is useful in breaking ties for consistent behavior across runs.

All other netlist functionalities are tool-specific, e.g., getting the timing/position of a pin. Note that the above netlist behavior does not require a particular netlist representation.

We use a *netlist adaptor* to capture the required behavior of netlists. A netlist adaptor is declared as a C++ abstract class with only pure virtual functions; implemented by `interact`, the coordinator of our tools; and then provided to gate-level tools upon their creation by `interact`. Individual tools can use the netlist adaptor at their discretion, for instance:

- Map ACT netlist objects to their internal objects, and vice versa.
- Facilitate more understandable debug messages.
- Perform file inputs and outputs.

Netlist objects are communicated among tools as `void*` for trivial copying, hiding netlist types, and decoupling of tools.

Using our netlist adaptor, a tool can represent a netlist based on its needs without interfering with other tools. For example, a placer can represent cell instances as nodes and nets as edges, while a timer can represent pins as nodes and timing arcs/net legs as edges. Furthermore, all gate-level tools can work with any netlist representation by replacing the netlist adaptor being provided. This greatly enhances the modularity of tools. Finally, full names for netlist objects are always consistent across tools, as they are centrally managed by a netlist adaptor.

Our design of netlist adaptors is an example of *dependency inversion* principle in software engineering. Without a netlist adaptor, gate-level tools depend on a given netlist representation. After introducing a netlist adaptor, however, a netlist representation now implements an abstract netlist adaptor, while all gate-level tools just use a netlist adaptor instance.

IV. Timer APIs

Since individual tools can have their own netlist representation, communication among tools is achieved through API calls mediated by `interact`. Any design optimizer relies on slack to know where to focus, and uses paths with delay annotated to decide what to do. Therefore, a timer should be able to provide slack at pins, and paths related to the slack value. Our Cyclone timer provides correctness slack from timing constraints, and performance slack from the critical cycle [5]. Since a pin may be involved in multiple timing constraints, a way to iterate over timing constraints should be provided by a timer.

A timer should be able to update timing incrementally by taking the manipulation results from circuit optimizers. For gate sizing and buffering, this is the change of netlists; for placement and routing, this is the change of RC trees for nets. Therefore, a timer should allow tools to propagate updates in netlist topology, attributes of netlist objects, and RC trees.

To support timing-driven design optimization, the *PhyDB* library has a set of callback functions that can be used by any of the back-end tools to issue timer queries. `interact` provides implementations of the callback functions that can issue the appropriate timer queries, whose results can be interpreted by the back-end tools using internal maps between

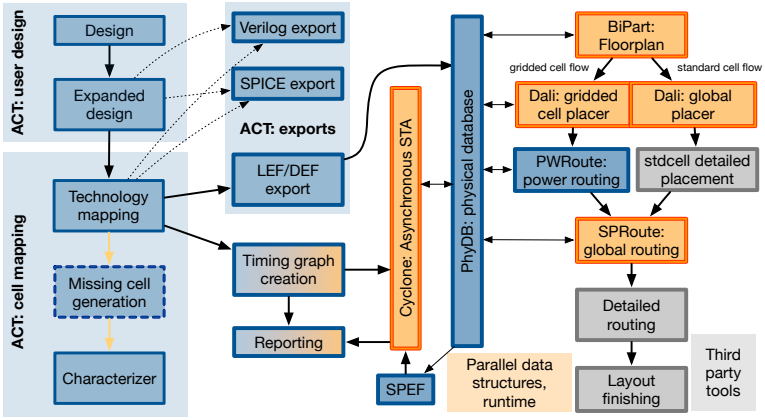

Fig. 1: `interact` components and interactions.

tool-specific netlist information (e.g., a *PhyDB* net) and the "master" ACT netlist information.

V. Supporting third-party flows

The design flow is specified via scripts in the `interact` tool. Hence, it is possible to customize the flow to support designs that were either hand-generated or generated through third party tools. We describe how this is supported in the current version of `interact`.

We assume that third-party designs are provided in a Verilog netlist. The ACT tools come with a program that can automatically convert a Verilog netlist into ACT syntax. Hence, it is this converted ACT file that serves as the entry point to our flow. This import also requires blackbox ACT declarations for all the cells in the standard cell library used by the original Verilog netlist.

The design can be read into `interact` via the automatically generated ACT netlist file. For static timing analysis, asynchronous circuits need additional information to correctly analyze cycles in the timing graph [5]. We have augmented `interact` to include commands to specify this information directly, as it will be missing from a plain Verilog netlist. `interact` also has commands to augment the set of timing constraints needed for correct operation. If required, these must also be specified via scripts because the Verilog netlist will not include them. Note that in a design that used ACT for design entry (prior to netlist generation), this information can be computed or specified in ACT, and is part of the timing graph generation algorithms [5].

To start the physical design, an external LEF must be provided for all the cells and technology design rules. The DEF is auto-generated from the ACT netlist, and the Dali global placer can be used. Dali also includes a gridded cell detailed placer. For standard cell placement, we support running an external standard cell detailed placer. Finally, we can run our global router and export LEF/DEF and guide files to run an open-source detailed router. Figure 1 provides an overview of our current flow. For shorter turn-around time, tools colored in orange are parallelized using the Galois framework, a C++ library for parallel graph computing [7] based on the operator formulation of algorithms [8]. Galois is available at [2].

VI. Current Status

We have used the current version of the flow to successfully re-implement a 65nm asynchronous microprocessor that we had previously taped out. Instead of using a collection of tools and scripts, all the ACT tools were executed directly via a single `interact` script prior to hand-off to commercial tools for detailed routing. `interact` is available at `http://github.com/asyncvlsi/interact`, and the components implemented as libraries are available at `http://github.com/asyncvlsi/`.

VII. Summary

We have described recent progress toward the creation of an open-source design flow for asynchronous logic. The main step in the past year has been to integrate all our core EDA tools into the `interact` interactive design environment. The flexibilty afforded by this integration means that we can support not just our ACT-specific flow, but also support third-party inputs using Verilog netlists, LEF, and scripts to specify additional timing constraints.

A true timing-driven flow is currently being implemented, and we hope to complete the first version of this flow by the end of this year.

References

[1] https://llvm.org/.

[2] Galois: C++ library for multi-core and multi-node parallelization. https://github.com/IntelligentSoftwareSystems/Galois.

[3] J. He, M. Burtscher, R. Manohar, and K. Pingali. Sproute: A scalable parallel negotiation-based global router. In *International Conference on Computer-Aided Design*, Nov. 2019.

[4] J. He, Y. Yang, and R. Manohar. A power router for gridded cell placement. In *Workshop on Open-Source EDA Technology, International Conference on Computer-Aided Design (ICCAD)*, Nov 2020.

[5] W. Hua, Y.-S. Lu, K. Pingali, and R. Manohar. Cyclone: A static timing and power engine for asynchronous circuits. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2020.

[6] S. Maleki, U. Agarwal, M. Burtscher, and K. Pingali. Bipart: A parallel and deterministic hypergraph partitioner. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 161–174, New York, NY, USA, 2021. Association for Computing Machinery.

[7] D. Nguyen, A. Lenharth, and K. 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.

[8] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI 2011*, pages 12–25, 2011.

[9] T. Spyrou. Opendb, openroad's database. In *Workshop on Open-Source EDA Technology (WOSET)*, 2019.

[10] Y. Yang, J. He, and R. Manohar. Dali: A gridded cell placement flow. In *International Conference on Computer-Aided Design (ICCAD)*, Nov 2020.