An Open-Source EDA flow for Asynchronous Logic

Samira Ataei, Wenmian Hua, Yihang Yang, Rajit Manohar *Computer Systems Lab*
Yale University
New Haven, CT, USA
Yi-Shan Lu, Jiayuan He, Sepideh Maleki, Keshav Pingali *Department of Computer Science*
University of Texas at Austin
Austin, TX, USA

***Abstract*—We present an open-source EDA flow for digital asynchronous circuits. One of the unique aspects of the flow is that it is designed so that it can support many different asynchronous circuit families in a unified framework, enabling the design of circuits that can leverage multiple circuit families and mix asynchronous and synchronous logic. The tools we are developing are built on top of the Galois parallel programming framework, which provides a new way to express algorithms that enables it to exploit irregular parallelism.**

***Index Terms*—asynchronous circuits; open-source EDA; parallel EDA tools**

I. Introduction

Scalable computer systems are designed as a collection of modular components that communicate through well-defined interfaces. The interfaces must be robust to delays and uncertainty in the physical implementation of communication. This view applies to computer systems at many levels of abstraction. The Internet is a collection of communicating computers with message-passing through the Internet protocol. A modern datacenter is a collection of servers that communicate via message-passing over commodity network hardware. Even large software systems consist of a collection of modules that use well-defined application programming interfaces (APIs) to communicate. Almost all computer systems disciplines have made the wise choice to partition their problem into components that communicate via protocols that *are independent of their physical realization—such as timing, energy, or size.*

The research described in this paper was supported by the DARPA IDEA program under contract FA8650-18-2-7850.

However, in current chip designs, this modular approach is abandoned in favor of global synchrony. A global synchronization signal (the "clock") dictates the time budget for every step of the computation—regardless of *what* is being computed.

Although this clocked design paradigm dominates the design of computers today, engineers are struggling to preserve the fiction of simultaneity required by the clock, even within an individual chip. This struggle is an inevitable result of advancing technology. As transistors get smaller and faster, the delay of communication over wires dominates the cost of local computation with transistors. Such progress renders the clocked paradigm a poorer and poorer abstraction for chip design. Modern application-specific integrated chips (ASICs) are designed as a collection of small clocked "islands" that communicate via interfaces that break the clocking abstraction.

To address this challenge, we are creating a collection of open-source electronic design automation (EDA) tools that isolate the designer from the details of the physical implementation technology, especially when it comes to delays and timing uncertainty.[1] The approach is based on an *asynchronous*, *modular* and *hierarchical* design methodology for complex chips, and it permits component re-use from one technology to another with little or no modification. While individual modules of the chip can be clocked, the overall system uses an asynchronous integration approach to achieve

[1]It is not possible to entirely decouple the logical correctness of a design from timing to create completely delay-insensitive circuits [1], [2]. However, it is possible to make a very mild and local timing assumption that is easy to satisfy in practice [3].

modular composition. Hence, the EDA flow being developed supports a combination of timing styles in an integrated framework.

The algorithmic complexity of some of the important steps in an EDA flow are higher when analyzing asynchronous circuits, which can have a major impact on the overall runtime of the flow. To reduce the turn-around time for designs, we are implementing parallel versions of the key algorithms in the tool chain, using the Galois system described in Section III. The Galois system supports parallelization of irregular algorithms such as those in which the key data structures are graphs and hyper-graphs. Since circuits can be viewed as hyper-graphs, the Galois system is well-suited for this parallelization effort.

II. Asynchronous logic: a unified approach

There are a large number of different asynchronous logic families. Historically, each of these families were developed by different research groups, with differing terminology and design methodologies. Note that this is not that different from synchronous logic; any textbook on synchronous digital logic will describe a large number of options for synchronous design such as pseudo-nMOS logic, pre-charge logic, dual-rail domino logic, and self-resetting logic to name just a few options [4]. In addition, many circuit options for flip-flops and latches are also described, and the merits of each discussed. Heterogeneity of this nature is difficult to incorporate into any automated design flow.

Instead, over the years, the mainstream industrial-strength ASIC flow that is provided by the major EDA vendors converged on a core synchronous EDA flow that supported a limited set of options. Today those options include flip-flop and (some) latch-based designs with excellent support for single-clock designs, and limited support for heterochronous designs. Circuit options were ignored in favor of standard-cell libraries with hand-optimized circuit layout for individual cells (an individual CMOS gate or a small collection of gates). This push was driven by industry, and resulted in standardization of what is viewed as the commercially supported ASIC flow today. Standardization led to interoperability, and a rich intellectual property (IP) ecosystem. Modern ASIC design is as much about system integration as it is about writing the detailed hardware description language that describes the chip.

The same cannot be said about asynchronous logic. The convergence that occurred in the synchronous domain did not occur in the asynchronous logic domain, and hence the EDA landscape for asynchronous logic is quite bleak in comparison. While there have been many academic tools developed for individual steps needed to go from a high-level description of an asynchronous design to a chip implementation, only a small number of complete flows have been developed. Example flows developed for asynchronous design include Haste [5] and Balsa [6], and more recently Proteus [7]. Each of these flows supports a restricted style of asynchronous design, and uses commercial synchronous tools for physical design automation. Since the synchronous physical design tools do not have a correct view of timing for asynchronous logic, conservative work-arounds are used to constrain the design in order to ensure a valid implementation. This approach has also been adopted by many academic groups to leverage the investment made by the commercial EDA industry.

A. A unified approach to asynchronous logic

Instead of developing an asynchronous logic flow that only supports a particular flavor of asynchronous logic (a survey of some approaches can be found in [8], [9], [10]), the approach we adopt tackles the problem of heterogeneity of design styles and circuit approaches.

1) Commonalities: Since one of the goals of adopting an asynchronous approach is to create a modular design style, all the differing approaches generally share the characteristic that a design is partitioned into a collection of concurrently operating hardware modules (we call them *processes*, adopting the term from the concurrent computing literature) that communicate with each other using well-defined protocols on wire bundles (we call the bundles *channels*). Channels are used both for exchanging information as well as synchronization between processes. To support this abstraction, we use a hardware specification notation called CHP (for Communicating Hardware Processes), an extension of Hoare's CSP language [11]. This is the

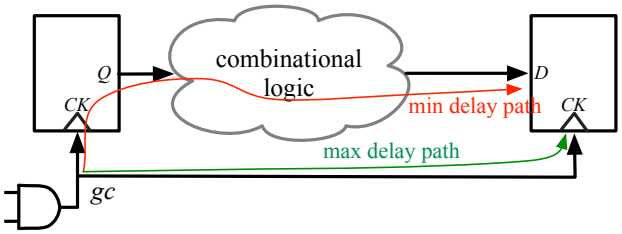

Fig. 1: Hold-time point-of-divergence constraint for generated clock *gc*.

highest level of abstraction, and corresponds to a behavioral description of the asynchronous chip. A CHP description can be translated into different asynchronous logic families.

2) Differences: The origin of a large number of differences between varying approaches to asynchronous design stems from differing assumptions about timing. Purely delay-insensitive circuits make no assumptions about the delays of gates or wires. This is an extremely robust approach, and requires at least two-output gates to be expressive enough [12]. No timing constraints have to be specified in this case. Quasi delay-insensitive circuits and speed-independent circuits require a timing assumption called the isochronic fork [2], which translates to a wire delay versus path delay assumption [3]. Bundled-data communication protocols require one wire (the request) to be slower than the data wires.

Even though there are a large range of timing requirements, many of them can be expressed using a generalization of two approaches to specifying timing constraints. The first is the approach used by synchronous timers to express hold time constraints for generated clocks. If a signal *gc* is a generated clock, and it is connected to two flip-flops, then the hold time constraint for the generated clock is a "point of divergence" constraint, where—starting from a root point—the maximum delay through one path has to be slower than the minimum delay through another path (see Figure 1). If the actual delay values are known, then `.sdc` constraints like `set_min_delay` and `set_max_delay` can be used to constrain the maximum and minimum delay on two paths so that they are ordered as required by the hold time constraint [13].

A second approach used by the asynchronous design community is generalized relative timing [14]. In this approach, constraints are specified on *signal transition events*. Hence, a point-of-divergence constraint would be expressed by using $gc\uparrow$ as the anchor event, and then using signal transitions at the input to the flip-flop from Figure 1. The challenge with using events (unlike paths in the synchronous case) is that events might have data-dependent occurrences. Also, if a signal can have switching hazards, it might result in more than one event corresponding to the change in the point of divergence.

To address these issues, we introduce a general version of both these notions that we call *timing forks*. A timing fork resembles a point-of-divergence constraint, except that it need not be a point of divergence. A timing fork `a+ : b- < c+` is a constraint that specifies a *error predicate*. In any execution of the circuit, if the sequence `a+` followed by `c+` followed by `b-` occurs *without an intervening* `a+`, then the constraint is violated. This timing fork is based on recent research in the distributed systems literature, that argues that events can be ordered only if there is a visible set of timing forks [15]. For isochronic forks, we need a notion of the near and far end of a wire; we augment the syntax of timing forks so that we can specify an input rather than the output of a gate. It should be clear that timing forks can express point-of-divergence constraints in a straightforward fashion. What is less obvious is that even when there isn't a local point-of-divergence, one must exist at some point in the execution history in the absence of any absolute notion of time [15].

Bundled data asynchronous communication uses a request line and a data bundle, where the data bundle signals have to be stable once the request goes high in the most straightforward version of the protocol. The communication is initiated by control logic; suppose that `c` goes high to initiate the communication, the request signal is `req`, and `di` is one of the data signals. The timing constraint would be written `c+ : di < req+`. Note that a scenario where `di` does not change is also permitted by the meaning of a timing fork. We distinguish between multiple uses of the same set of bundled data wires because `c+` will occur between each use of the wires.

The synchronous timing constraint shown in Figure 1 can be expressed using this notation in the following way: "`gc+ : FF.CK+ < FF.D`". This states that any changes in the `D` pin of `FF` must occur after the `CK` pin goes high, where `FF` is the

instance name of the flip-flop on the right hand side in Figure 1.

Since our timing constraints can specify both synchronous and asynchronous timing constraints, the flow supports a design with a mixture of synchronous and asynchronous components. In particular, timing forks are sufficiently expressive to describe the timing constraints needed for quasi delay-insensitive circuits, GasP pipelines [16], bundled-data communication, and high-speed transition signaling pipelines [17] to name a few circuit families.

B. The ACT framework

The flow we have developed includes a design language called ACT (for asynchronous circuit toolkit). ACT is a hierarchical design language that includes communication channels and encoded data values as first-class objects. The language supports representing circuits at multiple levels of abstraction, including CHP, gate-level, and transistor-level descriptions. ACT is strongly typed, and the type system is used to track and specify many design constraints that traditionally are externally specified in commercial flows (e.g. using `.sdc` files). Timing forks can be included as part of the logic specification.

By using an integrated language that can be used at multiple levels of abstraction, we preserve the relationships between different levels of abstraction in the design throughout the design flow. These relationships are captured using ACT's type system. Timing constraints between modules can be specified in the type-definition of communication channels—i.e. in the interface specification captured by the type signature of a component. 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. Hence, constraints generated by logic synthesis are made available to the rest of the flow as part of the ACT language, and hence are visible to both timing analysis and place-and-route tools.

The history of this language can be traced to the MiniMIPS project at Caltech (1994–1999), where a simplified version of ACT was developed by Manohar to manage the design complexity of the MiniMIPS asynchronous processor design [18]. This language, called CAST (for Caltech Asynchronous Synthesis Tools), was used to implement a microprocessor at the gate level of abstraction. This language was used both by Manohar's group at Cornell as well as a startup company (Fulcrum Microsystems). CAST continued to evolve at Fulcrum Microsystems, which was eventually acquired by Intel in 2012; as part of their development, Fulcrum also developed the Proteus flow [7]. The ACT language was created in 2005 as an evolution of CAST, and to overcome some of its limitations. This language was also used by Achronix Semiconductor, and to develop a number of chips at Cornell and Yale. An open-source version of this early version of ACT was also released [19]; these early versions of ACT were only designed to support quasi delay-insensitive asynchronous circuits.

The current ACT language [20] 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 [21] to 5.4B transistors [22], and in technologies ranging from 0.6μm CMOS to 28nm CMOS. It is general enough to be able to express a wide range of asynchronous logic families, and is the basis for the open-source EDA flow described in Section IV.

C. Keeping designs open

Our implementation of the ACT framework includes a number of configuration files. We have partitioned these files into two disjoint sets: technology-independent, and technology-specific. The information in the technology-specific files correspond to items that may be covered by non-disclosure agreements with foundries, and thus may not be distributed without the appropriate agreements in place. However, the goal is to ensure that the vast majority of the design can be distributed without reference to the details of the underlying technology.

Hence, while some of our tools cannot operate without detailed information from the foundry, the original logical design specified in the ACT framework can be distributed without any embedded foundry-specific information because this information is isolated to an input configuration file.

D. Integrating synchronous logic and other tools

The ACT tools are focused on supporting asynchronous logic families. However, we expect that a complex system would require integration with

other logic styles—in particular, synchronous logic. Since commercial EDA tools provide outstanding support for design styles commonly used by industry, we do not directly target synchronous logic in the ACT framework. Instead, we provide support for importing a design as a Verilog netlist. Importing synchronous logic or asynchronous logic from other flows into the ACT framework also requires that timing constraints for imported signals be specified using timing forks.

Module-level integration is the most straightforward way to import a Verilog netlist; this also permits a designer from maximizing the use of mainstream tools. Our timing analysis engine will determine that the synchronous logic and asynchronous logic are in different timing domains [23], and simply report timing for the different components separately. This is analogous to the scenario of an unrelated clock domain crossing that can occur in conventional EDA tools.

Asynchronous logic generated by any other approach can also be imported as long as the design is specified as a Verilog netlist, and the required timing constraints specified as timing forks. The ACT language also supports direct specification of logic using gates specified as pull-up and pull-down networks, bypassing the CHP level of circuit description and the Verilog import process.

III. PARALLELISM: THE GALOIS FRAMEWORK

Reducing the turn-around time of this design flow without sacrificing quality of results is critical for future designs. We believe this goal can be achieved by parallelizing the core EDA algorithms. Since circuits can be viewed abstractly as graphs and hyper-graphs, a system for supporting the design and implementation of a parallel EDA tool-chain must have the following characteristics.

- It must support clean abstractions for reasoning about and expressing the available parallelism in graph (and hyper-graph) algorithms.
- It must hide parallelization details such as synchronization from EDA algorithm designers.
- It must be scalable; as long as the algorithm has sufficient parallelism, performance should improve if more cores are used.

A. Operator formulation of algorithms

A clean abstraction for expressing parallelism in graph algorithms is the *operator formulation*, a *data-centric* abstraction in which algorithms are described as a composition of a *local view* and a *global view* of the computation.

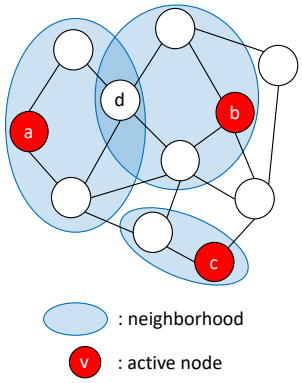

Fig. 2: Operator view of algorithms in Galois.

The local view is described by an *operator*, which is a graph update rule applied to an *active node* in the graph (some algorithms have active edges). Each operator application, called an *activity* or *action*, reads and writes a small region of the graph around the active node, called the *neighborhood* of that activity. Figure 2 shows active nodes as filled dots, and neighborhoods as clouds surrounding active nodes, for a generic algorithm.

An active node becomes inactive once the activity is completed. *Morph* operators can modify the graph structure of the neighborhood by adding and removing nodes and edges. And-inverter graph (AIG) rewriting [24] deploys morph operators. *Label computation* operators, in contrast, only update labels on nodes and edges without changing the graph structure. Field programmable gate array (FPGA) routing [25], formulated as a single-source shortest path problem (SSSP) within a routing resource graph, uses label computation operators.

The global view of a graph algorithm is captured by the location of active nodes and the order in which activities must appear to be performed. Topology-driven algorithms make a number of sweeps over the graph until some convergence criterion is met, e.g., the Bellman-Ford SSSP algorithm. Data-driven algorithms begin with an initial set of active nodes, and other nodes may become active on the fly when activities are executed. They terminate when there are no more active nodes. Dijkstra's SSSP algorithm is a data-driven algorithm. The second dimension of the global view of algorithms is *ordering* [26]. Activities in *unordered*

algorithms such as SSSP can be performed in any order without violating program semantics, although some orders may be more efficient than others.

Parallelism can be exploited by processing active nodes in parallel, subject to neighborhood and ordering constraints. The resulting parallelism is called *amorphous* data-parallelism. It is a generalization of the standard notion of data-parallelism [27].

B. Galois system

The Galois system implements this data-centric programming model (see details in [28]). 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 [29], FPGA routing [25], and AIG rewriting [24].

IV. The Open-source Flow

The key steps of the design flow we have developed are:

- Design elaboration/expansion, which expands the design and customizes it based on parameters specified by the user [20];
- Technology mapping and gate generation, which identifies the unique gates needed to implement the asynchronous circuit and generates the layout for the cells, if a new gate is found in the design [30], [31];
- Static timing analysis, which implements the asynchronous equivalent of timing analysis, determines the performance/power of the design, and checks any timing constraints needed for correctness [31];
- Design partitioning and floor-planning;

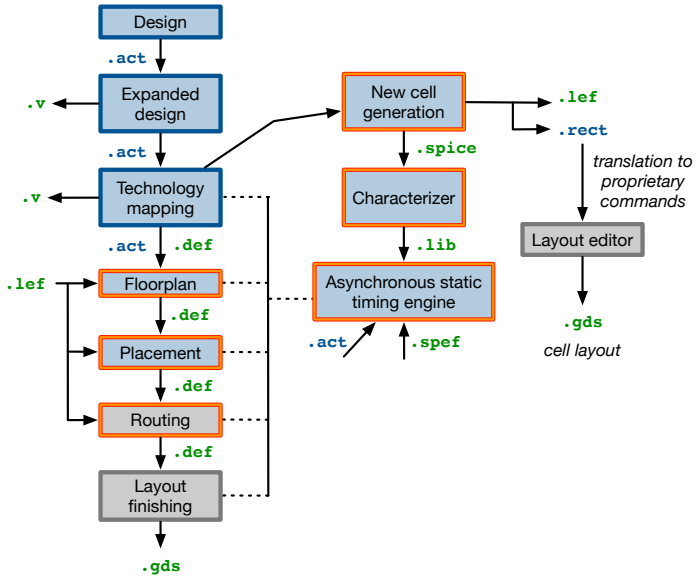

Fig. 3: Design flow for asynchronous logic.

- Asynchronous timing-driven placement [32], [33];
- Timing-driven global routing [34], followed by detailed routing to complete the physical implementation.

The rest of the steps are standard, including inserting fill and adding the pads and seal ring. The flow is summarized in Figure 3. To inter operate with commercial tools, at key steps we can import/export designs using standard formats such as a Verilog netlist, SPICE netlist, and LEF/DEF. We also accept parasitic information via SPEF files, and timing information using the `.lib` format. All the tools—both those under development and those ready for use—will be distributed at [35].

A. Timing and Power Analysis

We have implemented a static timing analysis engine for asynchronous logic. Since asynchronous logic might have non-standard gates, we also implemented a cell characterization engine that uses SPICE simulations to create `.lib` files for individual gates. The characterizer computes both delay and power tables for the gates.

Timing is one of the biggest differences between developing an asynchronous EDA flow and a synchronous EDA flow. Asynchronous timing has to handle cyclic circuit structures. Our timing analysis flow includes the following major steps: (i) creating an event-based timing graph for the asynchronous

TABLE I: Runtime of static timing analysis implemented using the Galois framework on a 56-core Intel Xeon server. The numbers in parentheses specify the number of threads used to obtain the best runtime. "bd203*" is a bundled-data sample design; the others are heavily pipelined and de-synchronized versions of their synchronous counterparts.

Name	#Pins	Seq (s)	Best(#thr) (s)	Speedup
bd203*	495	0.03	0.03 (1)	1.0
tv80-a	315,219	65.5	12.22 (42)	5.36
ac97ctrl-a	650,709	102.82	16.59 (28)	6.20
usbfunct-a	798,895	57.96	9.41 (42)	6.16
s38584-a	807,903	51.10	8.88 (35)	5.76
aescore-a	1,017,817	95.38	12.38 (42)	7.70
vgalcd-a	5,689,435	2,889.26	145.18 (56)	19.90

TABLE II: Performance comparison of our timing engine core against a recent open-source synchronous timing analysis engine (OT = OpenTimer). The numbers in parentheses specify the number of threads used to obtain the best runtime. All times are in milliseconds.

Circuit	# Pins	Best Runtime (#thr)		Speedup over OT
		OT	$Ours$	
ac97ctrl	40,238	312.0 (21)	35.3 (7)	8.83
aescore	66,221	493.3 (7)	53.0 (7)	9.31
desperf	295,808	2,762.7 (14)	155.0 (14)	17.82
vgalcd	380,730	3,660.7 (28)	187.7 (14)	19.51
desperf*10	2,958,071	29,923.3 (14)	1,366.7 (14)	21.90
vgalcd*10	3,807,291	31,212.7 (35)	1,708.7 (14)	18.27

design from a gate-level representation; (ii) estimating steady-state slew rates in cyclic circuits; (iii) analyzing the cycles in the event-based representation, and computing the critical cycle ratio which is a good metric of performance for the asynchronous circuit; (iv) computing arrival time and required time for asynchronous circuits, and hence computing the performance slack for each node of the timing graph; and (v) computing the slack for timing forks.

When importing a design from a different flow as a "black box," the timing graph fragment for the module must be included in the import (analogous to `.lib` files in synchronous logic). If a module is imported using a gate level netlist with timing forks, the event graph can be computed by our timing analysis engine.

Once event transitions are identified, we also compute the power consumption of the circuit. Since many of the steps needed by power analysis are the same as for timing analysis, we have integrated the two into a single unified engine.

The time complexity of timing analysis is much higher than in the synchronous case due to step (iii), which computes the critical cycle ratio. We use the parametric shortest path algorithm to compute this ratio [36], which provides better run-time performance than previous approaches that use linear programming when the circuit size is large [31]. Another source of complexity is that the periodicity of an asynchronous circuit may not be from one iteration to the next. Instead, a circuit might have an *unfolding factor* M, where the circuit timing is only periodic every M iterations [37], [38]. When M is high, timing propagation has to be performed on a graph that is logically the M-fold unfolding of the cyclic graph.

We have implemented our timing analysis engine using the Galois framework described in Section III. Our current implementation parallelizes the following parts of the full timing analysis: slew rate estimation, arrival time/required time, performance slack, and timing fork slack. The current runtime of our timing engine is shown in Table I with example circuits ranging from 0.3M–5.6M pins. The sequential runtime for large designs would be quite prohibitive—above 48 minutes for the largest example. However, the Galois framework can transparently speed up our runtime by large factors; for the largest example, we achieve almost 20× speedup, resulting in a more manageable runtime of roughly 2.5 minutes.

The same timing propagation core can be used to perform timing analysis for synchronous circuits, and hence, we can compare the performance obtained using the Galois approach to parallelization versus existing synchronous timing analysis engines that support multi-threaded execution. Table II shows the result of this comparison against OpenTimer, an open-source synchronous timing analysis engine that supports multi-threaded execution [39], demonstrating that our parallel timing analysis core achieves good parallel performance.

B. *Partitioning and floorplanning*

For large designs, we have implemented a min-cut based approach to floorplanning and design partitioning. To this end, we have developed a

deterministic, parallel hypergraph-based partitioner using the Galois framework.

Our implementation uses the multi-level graph partitioning framework, where the original hypergraph is subjected to a number of *coarsening* steps to create a much smaller hypergraph. The small hypergraph is then subjected to an initial partitioning. Finally, the small hypergraph is expanded out to the original graph by inverting the coarsening steps, and in each step the partition is further refined.

Experimental results on a 28-core Intel Xeon show that our partitioner achieves $7\times$ speedup for hypergraphs with roughly 10 million nodes and 4–$6\times$ speedup for hypergraphs with 2–3 million nodes.

C. Placement

Asynchronous circuits make use of a wide range of gates, especially state-holding gates that have pull-up and pull-down networks that are not complementary. The unbalanced pull-up and pull-down combined with keeper circuits for state-holding gates can lead to inefficient layout using traditional single/double height standard cells. To alleviate this potential inefficiency, we have adapted existing standard-cell based placement algorithms to account for cell heights that need not be uniform. We call this approach *gridded-cell* layout, where cells can have both height and width that is an integer multiple of a routing track. [32] Many techniques have been adapted for this non-standard height cell layout approach, and new algorithms developed for both fast legalization and well-alignment in the presence of non-uniform cell heights.

Experimental results show that our current placer implementation is capable of handling large designs, with a performance that is almost an order of magnitude faster than commercial placers while suffering a 13% (geometric mean) quality loss when measured in terms of half-perimeter wire length.

When comparing standard cell versus non-standard height cells for asynchronous logic, we found that our placement approach can improve density by 10% to 17% compared to commercial standard-cell placers [32].

D. Global routing

The last major piece of the flow that we are developing is a parallel global router. We have developed SPRoute, a Galois-based parallel implementation of the FastRoute [40] global router. We use FastRoute, because it has good sequential performance for the global routing problem.

Fig. 4: Routed example design in a 65nm process. Detailed routing was performed using a commercial EDA tool.

SPRoute uses a novel two-phase parallel scheme to achieve good speedup. In the initial phase, SPRoute exploits net-level parallelism. In this approach, different nets are routed in parallel. This proceeds until there is congestion due to a lack of routing resources. This phase can achieve significant speedup, because uncongested regions do not have any resource conflicts and thus net routing can proceed in parallel. Once congestion is detected, SPRoute switches to fine-grained parallelism where parallelism is exploited for frontier exploration during maze routing.

This scheme achieves $11\times$ speedup with 0.6% quality of results reduction on a 28-core machine when compared to the baseline FastRoute implementation [34].

After global routing, the rest of the flow can proceed using standard tools since all the major decisions that impact timing have been accounted for during placement and global routing. Figure 4 shows the routed design of a simple asynchronous benchmark circuit, where the detailed router used was a commercial tool. Note that the placement does not use the standard cell rows.

E. Memory compiler

The last major missing ingredient is a high quality memory compiler. Almost every digital ASIC requires memory, and asynchronous designs are no different. While many commercial memories include self-timed internal access, standard memory compilers provide a "black box" implementation that only provides a synchronous interface.

To address this challenge, we have built AMC, an asynchronous memory compiler [41] that is based on the OpenRAM framework [42]. AMC makes a number of changes to the design of the SRAM it generates compared to its baseline OpenRAM implementation: (i) it uses asynchronous logic to implement the control, and therefore provides an asynchronous interface to the core memory; (ii) it supports pipelined memory access for multi-banked memories with multiple in-flight transactions, where bank access is interleaved to improve effective throughput; (iii) it supports sub-banking with a hierarchical word-line structure to improve access time and reduce power consumption; (iv) it supports technologies upto 28nm, including thin cell and foundry cell bit-cells; (v) it supports an atomic read-modify-write operation, which takes significantly less time than a read followed by a write. AMC includes a built-in self-test engine, as well as synchronous wrapper circuits (at reduced performance).

Our comparisons show that the memories generated by AMC are competitive with published designs in the literature, as well as the memories available from the foundry [41].

F. Current status

We currently have a flow that can can be used to design and implement quasi delay-insensitive asynchronous circuits, as well as a restricted set of bundled-data circuits. The memory compiler has successfully been used to build memories in 65nm, 28nm, and 12nm process technologies, as well as older technology nodes. The full flow has been exercised to design a mixed quasi delay-insensitive/bundled data 65nm ASIC, and a 28nm ASIC is in progress. The key additional work needed to support a richer class of asynchronous circuits is a more general timing analysis engine front-end. While the core timing analysis engine implements the algorithms needed for asynchronous timing analysis, the front-end that generates the input for the analysis engine is currently being improved so that it can support a richer set of asynchronous circuit families. The rest of the physical design flow supports any asynchronous logic family. Finally, we are working on a tighter integration of the timing analysis engine with all the steps in the design flow.

V. Summary

We have embarked on developing a high quality open-source design flow for asynchronous circuits. In doing so, we developed a unified timing methodology that can handle both synchronous and a number of different asynchronous circuit families. By building this timing abstraction into all the key EDA tools, our goal is to create an extensible framework where EDA developers can easily support new circuit families.

Significant work is still required both for improving the run-time performance of certain aspects of the flow, as well as improving the quality of results of the design. Some of the major ongoing efforts include: improving the accuracy of timing analysis, as well as its run-time performance in the presence of millions of timing constraints; better incorporation of timing information into both placement and routing, as well as buffer insertion when timing constraints cannot be met during place and route; improved cell generation when a circuit is not found in the standard library; and extending configuration files and algorithms to support the requirements of sub-10nm designs.

References

[1] R. Manohar and Y. Moses, "The eventual c-element theorem for delay-insensitive asynchronous circuits," in *Asynchronous Circuits and Systems (ASYNC), 2017 23rd IEEE International Symposium on*, pp. 102–109, IEEE, 2017.

[2] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Sixth MIT Conference on Advanced Research in VLSI* (W. J. Dally, ed.), pp. 263–278, 1990.

[3] R. Manohar and Y. Moses, "Analyzing isochronic forks with potential causality," in *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pp. 69–76, IEEE, 2015.

[4] N. H. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective*. Pearson, 2010.

[5] S. F. Nielsen, J. Sparso, J. B. Jensen, and J. S. R. Nielsen, "A behavioral synthesis frontend to the haste/tide design flow," in *Asynchronous Circuits and Systems, 2009. ASYNC'09. 15th IEEE Symposium on*, pp. 185–194, IEEE, 2009.

[6] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[7] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC flow for GHz asynchronous designs," *IEEE Design and Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[8] S. Hauck, "Asynchronous design methodologies: An overview," *Proceedings of IEEE*, vol. 83, pp. 69–93, Jan 1995.

[9] S. M. Nowick and M. Singh, "Asynchronous design—part 1: Overview and recent advances," *IEEE Design & Test*, vol. 32, no. 3, pp. 5–18, 2015.

[10] S. M. Nowick and M. Singh, "Asynchronous design—part 2: Systems and methodologies," *IEEE Design & Test*, vol. 32, no. 3, pp. 19–28, 2015.

[11] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[12] R. Manohar and Y. Moses, "Asynchronous signaling processes," in *Asynchronous Circuits and Systems (ASYNC), 2019 25rd IEEE International Symposium on*, IEEE, 2019.

[13] S. Gangadharan and S. Churiwala, *Constraining Designs for Synthesis and Timing Analysis*. Springer, 2013.

[14] S. A. Seshia, R. E. Bryant, and K. S. Stevens, "Modeling and verifying circuits using generalized relative timing," in *11th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 98–108, IEEE, 2005.

[15] A. Dan, R. Manohar, and Y. Moses, "On using time without clocks via zigzag causality," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pp. 241–250, ACM, 2017.

[16] I. Sutherland and S. Fairbanks, "Gasp: A minimal fifo control," in *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pp. 46–53, IEEE, 2001.

[17] M. Singh and S. M. Nowick, "Mousetrap: Ultra-high-speed transition-signaling asynchronous pipelines," in *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pp. 9–17, IEEE, 2001.

[18] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee, "The design of an asynchronous mips r3000 microprocessor," *Proceedings of the 17th Conference on Advanced Research in VLSI*, pp. 164–181, 1997.

[19] D. Fang. https://github.com/fangism/hackt/.

[20] R. Manohar. https://github.com/asyncvlsi/act/.

[21] C. T. O. Otero, J. Tse, R. Karmazin, B. Hill, and R. Manohar, "ULSNAP: An ultra-low power event-driven microcontroller for sensor network nodes," in *15th International Symposium on Quality Electronic Design*, pp. 667–674, IEEE, 2014.

[22] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, *et al.*, "Truenorth: Design and tool flow of a 65mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, October 2015.

[23] R. Manohar, "Exact timing analysis for asynchronous circuits with multiple periods," *To appear, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[24] V. Possani, Y.-S. Lu, A. Mishchenko, K. Pingali, R. Ribas, and A. Reis, "Unlocking fine-grain parallelism for aig rewriting," in *ICCAD '18: International Conference on Computer Aided Design*, 2018.

[25] Y. O. M. Moctar and P. Brisk, "Parallel fpga routing based on the operator formulation," in *DAC '14: Design Automation Conference*, 2014.

[26] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, (New York, NY, USA), pp. 3–12, ACM, 2011.

[27] 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*, pp. 12–25, 2011.

[28] 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, (New York, NY, USA), pp. 456–471, ACM, 2013.

[29] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Commun. ACM*, vol. 59, pp. 78–87, Apr. 2016.

[30] R. Karmazin, C. T. O. Otero, and R. Manohar, "celltk: Automated layout for asynchronous circuits with nonstandard cells," in *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pp. 58–66, IEEE, 2013.

[31] W. Hua, Y.-S. Lu, K. Pingali, and R. Manohar, "Cyclone: a static timing and power analysis engine for asynchronous circuits," in *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems*, May 2020.

[32] Y. Yang, J. He, and R. Manohar, "Dali: a gridded cell placement flow," in *IEEE International Conference on Computer-Aided Design*, Nov 2020.

[33] R. Karmazin, S. Longfield, C. T. O. Otero, and R. Manohar, "Timing driven placement for quasi delay-insensitive circuit," *Proceedings of 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 45–52, 2015.

[34] J. He, M. Burtscher, R. Manohar, and K. Pingali, "Sproute: A scalable parallel negotiation-based global router," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2019.

[35] S. Ataei, W. Hua, Y. Yang, Y.-S. Lu, J. He, S. Maleki, R. Manohar, and K. Pingali. https://github.com/asyncvlsi/.

[36] N. E. Young, R. E. Tarjan, and J. B. Orlin, "Faster parametric shortest path and minimum-balance algorithms," *Networks*, vol. 21, no. 2, pp. 205–221, 1991.

[37] W. Hua and R. Manohar, "Exact timing analysis for asynchronous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 203–216, 2018.

[38] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello, "An algorithm for exact bounds on the time separation of events in concurrent systems," Tech. Rep. 94-02-02, University of Washington, Department of Compter Science and Engineering, 1994.

[39] T.-W. Huang and D. F. M. Wong, "Opentimer: A high-performance timing analysis tool," in *ICCAD '15: International Conference on Computer Aided Design*, 2015.

[40] M. Pan, Y. Xu, Y. Zhang, and C. Chu, "Fastroute: An efficient and high-quality global router," *VLSI Design*, vol. 2012, 2012.

[41] S. Ataei and R. Manohar, "Amc: An asynchronous memory compiler," in *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 1–8, IEEE, 2019.

[42] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "Openram: An open-source memory compiler," in *Proc. 35th International Conference on Computer-Aided Design*, pp. 93:1–93:6, 2016.

PLACE PHOTO HERE

R ajit Manohar is the John C. Malone Professor of Electrical Engineering and Computer Science at Yale University, New Haven, CT, USA. His research interests are in the design and implementation of asynchronous circuits and systems. He has a Ph.D. in Computer Science from Caltech.

PLACE PHOTO HERE

K eshav Pingali is the W.A."Tex" Moncrief Chair of Computing and CEO of Katana Graph. He has a PhD from MIT. He is a Foreign Member of the Academia Europeana, a Distinguished Alumnus of IIT Kanpur, India, and a Fellow of the ACM, IEEE and AAAS.

PLACE PHOTO HERE

S amira Ataei is an Associate Research Scientist with Yale University. She received her Ph.D. degree in Electrical Engineering from the Oklahoma State University, OK, USA in 2017. Her research interests include memory design for end-of-the-roadmap silicon, memory compiler, in-memory/near-memory computing and computer architecture.

PLACE PHOTO HERE

W enmian Hua is currently with Synopsys Inc. He received his Ph.D. degree in Electrical and Computer Engineering from Cornell University, Ithaca, NY, USA in 2020. His research interests are in timing and performance analysis of asynchronous circuits.

PLACE PHOTO HERE

Y ihang Yang is currently pursuing his Ph.D. degree at Yale University. He received the MASc degree in electrical and computer engineering from the University of Waterloo in 2017. His research interests include asynchronous VLSI design and its physical design automation.

PLACE PHOTO HERE

Y i-Shan Lu is a PhD candidate in the Department of Computer Science, UT Austin. He received his Master degree in Computer Science from NTHU, Hsinchu, Taiwan in 2011. His current research focuses on parallelization and language design for domain specific computation, e.g., EDA. He is an ACM student member.

PLACE PHOTO HERE

M ichael Jiayuan He is currently a Computer Science PhD student at UT Austin. His research interests include parallel computing on multicore CPUs and GPUs, graph analytics and place & route in EDA. He received his B.E in Electrical Engineering and a second Bachelors in Economics from Tsinghua University in 2014.

PLACE PHOTO HERE

S epideh Maleki is a fifth-year Computer Science PhD student at the University of Texas at Austin. Her research interests are graph analytics, high performance computing, electronic design automation (EDA), and programming languages. She received her Masters in Computer Science from Texas State University.