



AFRL-RY-WP-TR-2024-0135

AN INTELLIGENT DESIGN ENVIRONMENT FOR ASYNCHRONOUS LOGIC (IDEAL)

**Rajit Manohar
Yale University**

**Keshav Pingali
University of Texas at Austin**

**Martin Burtscher
Texas State University**

**Prasad Joshi
Intel**

**Matthew Guthau
UC Santa Barbara**

**JUNE 2024
Final Report**

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

See additional restrictions described on inside pages

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC)
(<http://www.dtic.mil>).

AFRL-RY-WP-TR-2024-0135 HAS BEEN REVIEWED AND IS APPROVED FOR
PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//signature//

CHRISTOPHER A. BOZADA
Program Manager
Aerospace Components and Subsystems Division

//signature//

JOHN S. CETNAR (acting)
Deputy Chief, Aerospace Components &
Subsystems Technology Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show “//Signature//” stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE June 2024		2. REPORT TYPE Final		3. DATES COVERED	
				START DATE 13 June 2018	END DATE 15 September 2023
4. TITLE AND SUBTITLE AN INTELLIGENT DESIGN ENVIRONMENT FOR ASYNCHRONOUS LOGIC (IDEAL)					
5a. CONTRACT NUMBER FA8650-18-2-7850		5b. GRANT NUMBER N/A		5c. PROGRAM ELEMENT NUMBER 62716E	
5d. PROJECT NUMBER N/A		5e. TASK NUMBER N/A		5f. WORK UNIT NUMBER Y1TB	
6. AUTHOR(S) Rajit Manohar (Yale University) Keshav Pingali (University of Texas at Austin) Martin Burtscher (Texas State University) Prasad Joshi (Intel) Matthew Guthaus (UC Santa Barbara)					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Yale University 205 Prospect St, New Haven, CT 06511 University of Texas at Austin 1701 Red River St, Austin, TX 78701		Texas State University 601 University Dr, San Marcos, TX 78666 Intel 2200 Mission College Blvd, Santa Clara, CA 95054 University of California Santa Barbara Santa Barbara, CA 93106		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command United States Air Forces		Defense Advanced Research Projects Agency (DARPA/MTO) 675 North Randolph Street Arlington, VA 22203		10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RYPD	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-RY-WP-TR-2024-0135	
12. DISTRIBUTION/AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES This material is based on research sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7850. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), or the U.S. Government. Report contains color.					
14. ABSTRACT This project developed an open-source layout generator for the design and implementation of digital asynchronous circuits. To realize the goals of the Intelligent Design of Electronic Assets (IDEA) program for an open-source "no human in the loop" circuit layout generator, tools were created that support a modular and hierarchical design method that maximally decouples front-end design details and automation flows from proprietary foundry rules, models, and design kits. Individual modules can be designed using any methodology, but the global chip design adopts an asynchronous approach to integration and modular composition.					
15. SUBJECT TERMS asynchronous circuits, electronic design automation, layout generator, modular composition, application specific integrated circuits					
16. SECURITY CLASSIFICATION OF:				17. LIMITATION OF ABSTRACT	
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	SAR		18. NUMBER OF PAGES 68
19a. NAME OF RESPONSIBLE PERSON Christopher Bozada				19b. PHONE NUMBER (Include area code) N/A	

Table of Contents

Section	Page
List of Figures	iii
List of Tables	iv
1 INTRODUCTION	1
2 METHODS AND PROCEDURES	2
2.1 Asynchronous Design Principles and Its Relation to Design Automation	2
2.2 The ACT Hardware Description Language	4
2.3 The Galois Framework for Fine-grained Parallelism	6
2.4 Logic Synthesis and Cell Mapping	7
2.5 Parallel Partitioning	7
2.6 Gridded Cell Placement	8
2.7 Parallel Global Routing	10
2.8 Parallel Asynchronous Static Timing Analysis	11
2.9 Memory Compiler	12
2.10 Industry-standard File Exports	13
2.11 Overall Design Automation Flow	13
3 RESULTS AND DISCUSSION	15
3.1 Community	15
3.2 Chip Designs	16
3.3 Intel Study	16
3.3.1 Benchmarks	17
3.3.1.1 Multiply-accumulate	18
3.3.1.2 Floating-point multiply-add	19
3.3.1.3 Processor	21
3.3.1.4 Complex neuromorphic designs	22
3.3.1.5 Neuromorphic Core A	22
3.3.1.5.1 Neuromorphic Core B	23
3.3.1.6 Network-on-chip router	24
3.3.2 Placement	24
3.3.2.1 Evaluation Method	25
3.3.2.1.1 Half-perimeter Wire Length	27
3.3.2.1.2 Density	30
3.3.2.1.3 Routing Congestion	32
3.3.2.1.4 Timing Slack	36
3.3.2.2 Blockages and Placement Regions	40
3.3.2.3 Compute Resource Usage	42
3.3.2.4 Suggestions for Improvement	45
3.3.3 Static timing analysis	46
3.3.3.1 Bundled-data asynchronous design	46
3.3.3.1.2 Circuit timing constraints	48
3.3.3.1.3 Timing Graph and Ticks	49
3.3.3.1.4 Timing forks	50
3.3.3.1.5 False Timing Paths	52

Section	Page
3.3.3.1.6Future Directions	53
3.3.3.1.7Suggestions for Improvement.....	53
3.3.4 Summary	55
3.4 Transitions.....	55
3.5 Summary of Open-Source Software Tools Developed.....	56
4 CONCLUSIONS.....	58
5 REFERENCES	59
6 LIST OF ACRONYMS	61

List of Figures

Figure	Page
Figure 1. Four-phase Bundled-data Handshake Protocol	2
Figure 2. A linear Asynchronous Pipeline implemented with Quasi Delay-insensitive Control Logic	3
Figure 3. The Galois View of a Computation and its Operation on a Graph	6
Figure 4. Gridded Cells Compared to Standard Cells.....	9
Figure 5. The Gridded Cell Placement showing Mini-rows and Diffusion Alignment.....	10
Figure 6. Global Routing on a Coarse Grid	11
Figure 7. The Overall Implementation Flow Developed by the IDEA Project	14
Figure 8. Distribution of Registrations for the Summer School on Asynchronous Design.....	15
Figure 9: Multiply-accumulate CSP Circuit Description.....	18
Figure 10: Multiply-accumulate Dataflow Diagram.....	18
Figure 11: Multiply-accumulate [25%] Physical Placement Example (commercial flow)	19
Figure 12: Floating-point Multiply-add CSP Circuit Description	20
Figure 13: Floating-point Multiply-add Dataflow Diagram	20
Figure 14: Floating-point Multiply-add [60%] Physical Placement Example (commercial flow).....	21
Figure 15: Processor Physical Placement Example (commercial flow)	22
Figure 16: Neuromorphic Core A Physical Placement Example (commercial flow).....	23
Figure 17: Neuromorphic Core B Physical Placement Example (commercial flow).....	23
Figure 18: Router Physical Placement Example (commercial flow).....	24
Figure 19: Half-perimeter Wire Length Comparison for Floating-point Multiply-add [60%].....	27
Figure 20: Half-perimeter Wire Length Comparison for Neuromorphic Core A and Processor .	28
Figure 21: Half-perimeter Wire Length Comparison for All Designs.....	29
Figure 22: Multiply-accumulate [25%] – commercial vs Dali Placement	30
Figure 23: Multiply-accumulate [25%] – varying Dali Target Density	31
Figure 24: Floating-point Multiply-add [60%] - global Routing Congestion.....	33
Figure 25: Floating-point Multiply-add [60%] – placement with Congestion	35
Figure 26: Neuromorphic Core A - global Routing Congestion.....	36
Figure 27: Floating-point Multiply-add [60%] - timing Slack	37
Figure 28: Neuromorphic Core A and Processor - timing Slack	39
Figure 29: Neuromorphic Core B - layout with Non-rectangular Floorplan	41
Figure 30: Router Placement with Blockages and Regions.....	42
Figure 31: Bundled-data Channel Handshakes.....	47
Figure 32: Bundled-data FIFO Pipeline.....	47
Figure 33: Latch Timing Constraints.....	48
Figure 34: Timing Graph Tick Example.....	49
Figure 35: Setup, Hold, and Pulse Width Constraints Expressed as Timing Forks.....	51
Figure 36: False Timing Fork Paths in FIFO.....	52
Figure 37: Incorrectly Reported Timing Path.....	53

List of Tables

Table	Page
Table 1: Layout Cell Categories	17
Table 2: Multiply-accumulate [25%] - basic composition	19
Table 3: Floating-point Multiply-add [60%] - basic Composition	20
Table 4: Processor - basic Composition.....	21
Table 5: Neuromorphic Core A - basic Composition	22
Table 6: Neuromorphic Core B - basic Composition	23
Table 7: Router - basic Composition	24
Table 8: Key Quality of Results Figures of Merit	25
Table 9: Neuromorphic Core A and Processor - timing Slack Summary	39
Table 10: Placement Runtime Comparison	43
Table 11: Placement Memory Usage Comparison	44
Table 12: Compute Resource Usage Summary Comparison.....	45

1 INTRODUCTION

The goal of this effort was to create open-source electronic design automation (EDA) tools that enable the design of application-specific integrated circuits (ASICs) that use asynchronous logic. Design automation for asynchronous logic circuits presents several unique challenges: non-standard logic gates that are unsupported by commercially available libraries, cyclic circuit topologies that cannot be handled by standard timing analysis tools, and the lack of memory macros that are essential to any complex digital ASIC.

This project developed an open-source layout generator for the design and implementation of digital asynchronous circuits. To realize the goals of the Intelligent Design of Electronic Assets (IDEA) program of having an open-source “no human in the loop” circuit layout generator, we created tools that support a modular and hierarchical design method that maximally decouples front-end design details and automation flows from proprietary foundry rules, models, and design kits. Individual modules can be designed using any methodology, but the global chip design adopts an asynchronous approach to integration and modular composition. To keep the methodology and tools as open as possible, we adopted a generative approach to layout creation where our technology description is sufficient to create legal geometries (akin to the “scalable” rules from the 1980s), but not detailed enough to correspond to proprietary foundry rules.

Wherever possible, our project re-used software tools from the mainstream synchronous logic design flow. To support this philosophy, our software flow can inter-operate with third party open or closed source tools for steps such as extraction of parasitic resistance and capacitance values, layout-versus-schematic checking, design rule checking, detailed routing, and analog simulation.

Our tools were evaluated in several ways. First, we used our tools to tape-out a few different ASICs to validate our tools and show that they can be used to create physical chips that use asynchronous digital logic. Second, we collaborated with Intel’s asynchronous design group—the leading asynchronous design team in industry—to evaluate our tools and identify shortcomings and directions for future extensions. Finally, our tools have been used by numerous external users in both industry and academia to design asynchronous digital hardware.

2 METHODS AND PROCEDURES

2.1 Asynchronous Design Principles and Its Relation to Design Automation

The clocked design paradigm dominates the design of computers today. It relies on the simultaneity of events over physical space—a reliance that violates physics. Even within an individual chip we now see engineers struggling to preserve the fiction of simultaneity required by the clock. The 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 realize the IDEA program goals of completely decoupling front end design details and EDA flows from proprietary foundry rules, models, and design kits, it is essential to adopt a *modular, hierarchical* design methodology for complex System-on-a-Chips (SoCs). Such an approach not only permits component re-use from one technology to another with little/no modification, but it also permits the computer-aided design (CAD) flows to scale up to massively parallel computer platforms. Individual *modules* could be synchronous or asynchronous, analog or digital; but the overall system must use an asynchronous integration approach to achieve modular composition.

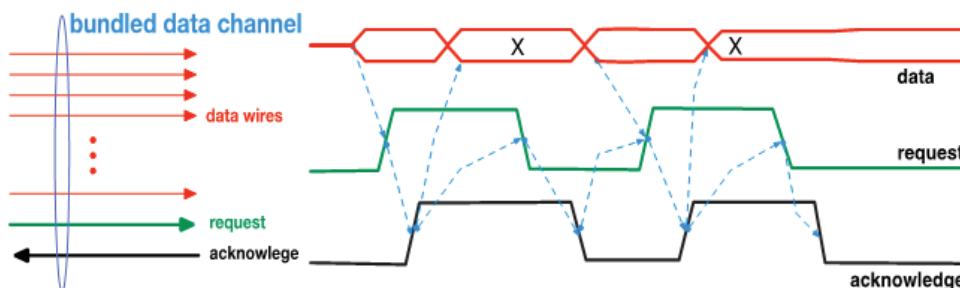


Figure 1. Four-phase Bundled-data Handshake Protocol

An asynchronous system is best described as a collection of communicating hardware *processes* that interact via first-in first-out message-passing *channels*. This corresponds closely to the software distributed systems model that has resulted in large-scale systems like the global Internet, and the asynchronous design literature borrows heavily from work in distributed software systems. The physical realization of channels is a collection of wires that implement *handshake protocols*. A commonly used wire realization is the *four-phase bundled data* handshake protocol, that uses two wires for synchronization and N wires for N-bit data communication. The two synchronization wires replace the “ready” and “valid” signals and shared clock used in synchronous communication protocols that use flow-control. The protocol is illustrated in Figure 1, and consists of four steps: (1) the sender sets the data bits, and changes the request signal via a zero-to-one transition; (2) the receiver reads the data bits and then sets the acknowledge wire (zero-to-one transition); (3) the sender then resets the request (one-to-zero transition); (4) the receiver then resets the acknowledge. This brings the control wires back to the initial state, and the channel is ready to send the next value. For this protocol to operate correctly,

a timing assumption is required—namely that the data wires are stable when the request changes from zero to one, and the data wires cannot be changed until the receiver sets the acknowledge high.

We now illustrate several different challenges that must be tackled when developing a *general* design automation flow for asynchronous logic.

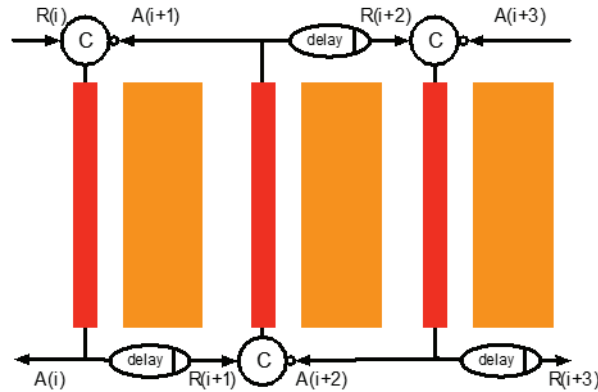


Figure 2. A linear Asynchronous Pipeline implemented with Quasi Delay-insensitive Control Logic

This circuit is called a micropipeline. The boxes in red are asynchronous registers that hold data, and the orange regions correspond to combinational logic. The control logic is shown in black.

Challenge: Variations on Channel Handshake Protocols: There are many useful variations on the specific handshake protocol describe above. Instead of four-phase handshakes, *two-phase* handshake protocols can be used. Instead of bundled data, the data wires themselves can be encoded using delay-insensitive codes eliminating the need for the timing assumption inherent in the bundled-data protocol. These are choices that should be permitted in any description of an asynchronous logic implementation flow.

Challenge: Hazard-free logic synthesis: The implementation of the logic of an individual process in an asynchronous system also differs from its synchronous counterpart. In standard clocked flip-flop based synchronous logic, the clock signal is used to determine *when* a signal value is sampled. In between clock edges, signal values can be incorrect, have glitches, or transition multiple times. This is acceptable so long as the signal value is correct in a small window around the sampling clock edge. In the asynchronous setting, the clock is absent. Hence, certain signals (e.g. the request signal in the four-phase bundled data protocol) must be *glitch-free* (a.k.a. hazard-free), as they are used to determine state machine transitions. This requires *hazard-free* logic synthesis, at least for some parts of the asynchronous logic implementation. Hence, asynchronous control logic requires more sophisticated logic synthesis algorithms.

Challenge: Many different timing assumptions: Another challenge in design automation is that there are several *useful* asynchronous logic families. The *quasi delay-insensitive* logic family [1]–[3] is extremely robust to variations in manufacturing process, operating voltage, and operating temperature. However, this comes at a cost in logic complexity (area). Pulsed logic

families like GasP [4] and MOUSETRAP [5] are examples of extremely high throughput pipelines, but they require additional timing assumptions to guarantee correct behavior. Hence, a general asynchronous layout generation flow must support a wide range of different timing requirements—and ideally be “future proof.”

Challenge: Non-standard state-holding gates: The circuits that are generated by logic synthesis for asynchronous design include gates that are typically not found in commercial standard cell libraries. Figure 2 shows a linear asynchronous pipeline consisting of micro-pipelined control logic. The control logic uses an inverting C-element, a standard *state-holding* gate that is widely used in asynchronous circuits, but that is absent from standard commercial cell libraries. Hence, an asynchronous layout generator must include a mechanism for a designer to create the new cells required for the asynchronous logic family of interest.

Challenge: Timing analysis for cycles of gates: Figure 2 also illustrates another challenge, namely that the circuit contains *cycles* of gates without any intervening register/flip-flop/latch. The oscillation period of these cycles determines the performance of the asynchronous circuit, and hence timing analysis must incorporate cyclic analysis.

We describe how we tackled the challenges outlined above in the Asynchronous Circuit Toolkit (ACT) system that we developed for the implementation of asynchronous logic.

2.2 The ACT Hardware Description Language

```
defproc fifo (chan?(int<8>) in; chan!(int<8>) out)
{
    int<8> x;
    chp {
        *[ in?x; out!x ]
    }
}
```

To support the range of channel protocols at an abstract level and to support the communicating process paradigm, we developed the ACT hardware description language. ACT contains processes and channels as first-class objects and can be used to describe complex asynchronous circuits at a high level of abstraction. As an illustrative example, the following describes a first-in first-out buffer that holds 8-bit integer data.

This defines the fifo process, with ports “in” and “out” that hold 8-bit integers. The process is specified in the Communicating Hardware Processes (CHP) syntax, which is widely used in the asynchronous design community. This syntax is based on Hoare’s CSP language [6] with a few extensions to support efficient hardware design. The “[“ and “]” brackets correspond to an infinite loop. The two operations in sequence in the body of the loop are “in?x” which receives the next data on the input channel “in” and stores it in variable “x”, and “out!x” which sends the current value of “x” on the output channel “out.”

To make this syntax a bit more accessible to users getting familiar with asynchronous design, we also have a more verbose notation called “chp-txt”, and the same process can be specified as shown below.

```
defproc fifo (chan?(int<8>) in; chan!(int<8>) out)
{
  int<8> x;
  chp-txt {
    forever {
      recv(in, x);
      send(out, x)
    }
  }
}
```

The documentation for the ACT language along with tutorials and videos were developed and are available on the ACT wiki that we actively maintain [7].

To support the range of timing assumptions in different asynchronous logic families, ACT supports the notion of a *timing fork*. A timing fork is a special way to express a signal transition ordering constraint. A timing fork is written as follows.

This constraint means the following: after signal “a” goes high, “d” cannot change until “b” goes

```
spec {
  timing a+ : d < b-
}
```

low. This constraint is powerful enough to support arbitrary asynchronous circuit families. In fact, our result results show that timing forks are necessary for signal transitions to be ordered in an asynchronous circuit setting [8].

To support non-standard gates, ACT permits the specification of arbitrary pull-up and pull-down networks as part of the hardware description language. The following describes the inverting C-element from Figure 2.

```
prs {
  a & b -> c-
  ~a & ~b -> c+
}
```

The syntax is called production rules (PRS) [7], and can be used to specify arbitrary switching pull-up and pull-down networks. The left-hand side is a Boolean equation, and the right-hand side is a signal transition. The example above behaves as follows: when “a” and “b” are both

high, “c” is pulled low; when “a” and “b” are both low, “c” is pulled high; in all other states, “c” holds its current state.

Hence, the ACT hardware description language directly supports standard asynchronous design abstractions and is expressive enough for designers to describe any special gates/logic they require for their circuits. In addition, complex timing constraints can be directly specified in the language as part of the requirements for the implementation.

2.3 The Galois Framework for Fine-grained Parallelism

As discussed above, we expect that design automation for asynchronous logic will have higher computational requirements compared to synchronous logic. To reduce the run-time in support of the IDEA goal of 24-hour turn-around time for chip implementation, we used parallel algorithms for several steps in the EDA flow: partitioning, global routing, placement, and timing analysis.

Since EDA algorithms operate on gates and their connections (nets), a graph or hypergraph is an appropriate representation of digital circuits for the implementation flow. We used the Galois system to simplify parallel implementation of graph algorithms [9]. The Galois system has been developed over almost two decades of effort by Prof. Keshav Pingali’s group at the University of Texas at Austin.

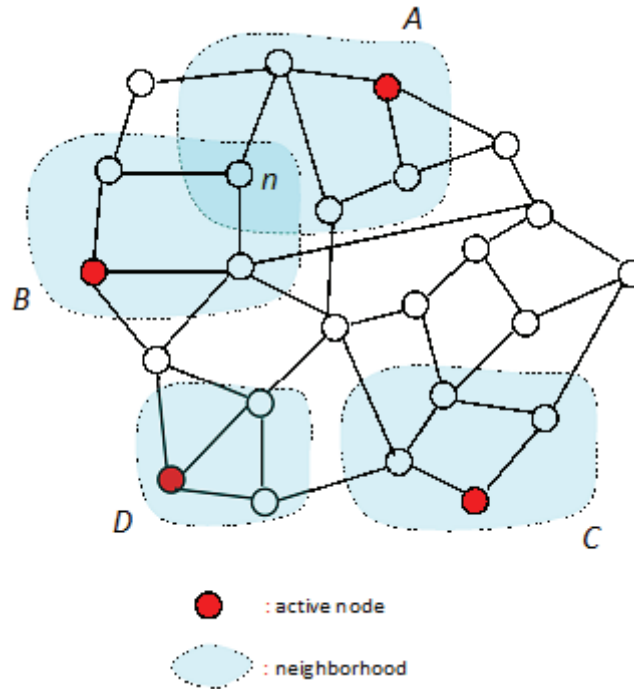


Figure 3. The Galois View of a Computation and its Operation on a Graph

The Galois system is an implementation of a *data-centric* programming model. 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 3 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. In general, operators can modify the graph structure of the neighborhood by adding and removing nodes and edges (these are called *morph* operators). In most graph applications, operators only update labels on nodes and edges, without changing the graph structure. These are called *label computation* operators. In the EDA context, timing analysis that computes the arrival time at each node would be an example of a label computation operator.

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 several sweeps over the graph until some convergence criterion is met. 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. The second dimension of the global view of algorithms is *ordering*. Many algorithms are *unordered* algorithms in which activities can be performed in any order without violating program semantics, although some orders may be more efficient than others. On the other hand, *ordered* algorithms require the active elements to be processed in an application-specific order for correct results.

Parallelism can be exploited by processing active nodes in parallel, subject to neighborhood and ordering constraints. The resulting parallelism is called *amorphous* data-parallelism, and it is a generalization of the standard notion of data-parallelism. Using the significant investment in performance tuning for unstructured graph-based parallelism that is available in the Galois system resulted in significant speedup for several different steps in our EDA flow.

2.4 Logic Synthesis and Cell Mapping

Asynchronous logic synthesis can result in non-standard gates. To support this, the ACT tools include a cell mapper (`prs2cells`) that extracts all the unique gates in the user's design. These unique gates are then automatically assembled into a *cell library*. Ideally the user would have the physical layout for each of the cells in the cell library; if so, the ACT implementation flow can use this directly. In the case when the design includes a novel cell, ACT also provides support to auto-generate part of the physical design of the novel cell. The cell layout must be completed by the user by completing the local cell wiring. This step is required only once for a particular technology and/or new logic family. ACT also provides a cell library characterization tool (`xcell`) that is used to compute timing summaries and power summaries for any new cells that were developed for the implementation flow [10].

Once the cells are ready, ACT can take a design description and automatically convert it into a layout place and route problem in industry-standard Library Exchange Format (LEF) and Design Exchange Format (DEF) format in an automated fashion (`act2lef`).

2.5 Parallel Partitioning

One of the first steps in a complex design is to partition the gates into regions and floorplan the design. The ACT tools implement a fast, parallel, hypergraph partitioning algorithm based on a new parallel implementation of the HMetis partitioner [11]. Our new partitioner, dubbed

“BiPart”, produces high quality partitions (as measured by the cut cost) with good runtime, because we can exploit fine-grained parallelism in Galois [12]. The table below shows an example comparison in run-time of BiPart compared to KaHyPar, a state-of-the-art hypergraph partitioner. For multi-way partitions, KaHyPar’s runtime is high enough that it did not complete after 30 minutes of runtime, whereas BiPart completes in under half a minute.

	BiPart		KaHyPar	
<i>K</i>	<i>Time (secs)</i>	<i>Edge Cut</i>	<i>Time (secs)</i>	<i>Edge Cut</i>
2	7.9	13,853	581.5	11,457
4	14.7	100,380	Timeout	-
8	17.5	185,079	Timeout	-
16	20.0	269,144	Timeout	-

2.6 Gridded Cell Placement

Given that the timing analysis needed for asynchronous circuits is quite different from clocked logic, we built a new placer from scratch to support asynchronous logic. In the process, we identified a key issue that made asynchronous row-based placement inefficient.

Asynchronous logic uses a wider range of cells compared to synchronous logic. As a result, there are gates with varying numbers of series transistors included that are not amenable to automated decomposition into smaller gates. In clocked logic, any combinational logic expression can be decomposed into two-input NAND gates, for example. This general decomposition is not possible in the asynchronous context; instead, gate decomposition is both state dependent and context dependent. Hence, the underlying circuit is often best implemented with complex state-holding logic. When the number of series transistors in a pull-up or pull-down network is increased, the physical width of the transistors must also be increased to preserve the drive strength of the network. This makes for “lopsided” n-transistor widths and p-transistor widths, depending on the specific gate type.

To make this conform to conventional standard cell-based layout, the maximum cell height across all gates would have to be increased compared to clocked circuits. In previous work, we found that 16 and 18 track height cells are common [13], while other cells (e.g. inverters used for delay lines) are very small but must be bloated to 16/18 tracks to conform to standard cell requirements and waste area.

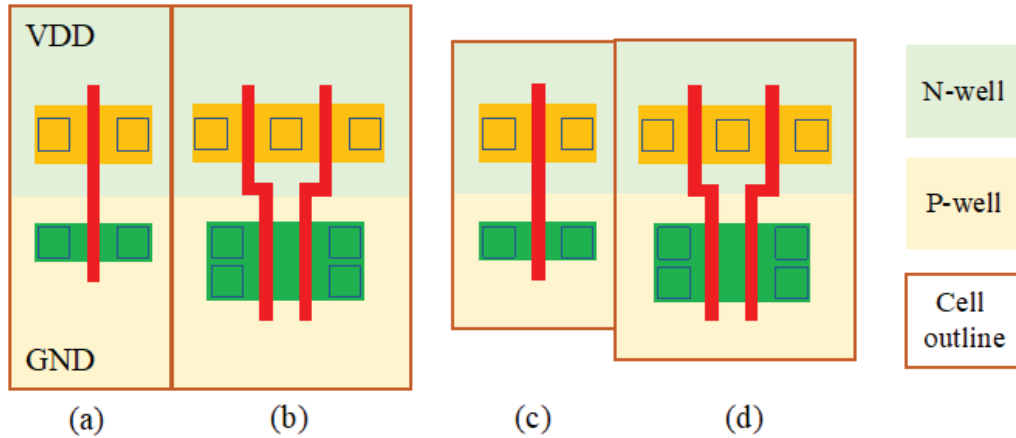


Figure 4. Gridded Cells Compared to Standard Cells

(a) and (b) show standard cells for an inverter and a two-input NAND gate respectively, (c) and (d) show the corresponding gridded cells. The gridded cells don't include their power stripes, and their cell outline can be an integer multiple of the routing grid in the vertical direction as well as the horizontal dimension.

We developed a *gridded cell* placement flow to circumvent this limitation. Instead of having a fixed track height (or just single and double height cells), our approach permits fine-grained cell heights. In the gridded approach, the cell height can be any integer multiple of the routing track. This is illustrated in Figure 4. To support this, a number of changes were made to the placement algorithm: (i) detailed placement must take the variable cell height into account; (ii) n-transistor and p-transistor diffusion/select regions must be aligned to avoid design rule errors; (iii) well regions must be drawn to account for variable height; and (iv) power stripes for power and ground distribution to the cells must take the variable height into account.

We solved this by sub-dividing the layout into *mini-rows*: each row is split into mini-rows of variable height. The height of a mini-row is the maximum of all the gridded cells placed within the mini-row. Figure 5 shows an example of a placement using the gridded cell approach. The right panel shows a closeup of a small section of the layout. The fact that the rows are not aligned across the entire horizontal section of the layout can be clearly seen.

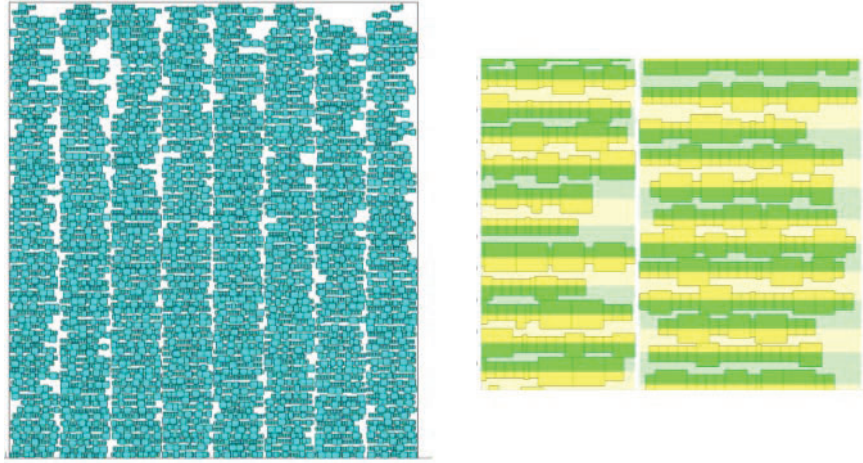


Figure 5. The Gridded Cell Placement showing Mini-rows and Diffusion Alignment
The green sections on the right correspond to p-type transistors and the yellow regions are the n-type transistors.

The placer we developed includes global placement, detailed placement, and final legalization in an integrated framework. In addition, the placer also supports standard cell placement. Placement steps can be customized, and the placement mode is a configuration option that is user settable.

The placer also includes an interface to the static timing analysis engine that we have developed (Section 2.8). The placer can find any timing constraint violations for the current placement and use this information to incrementally update the placement. The placer can also provide estimated parasitic resistance and capacitance values to the timing analysis engine and perform an incremental update of the current state of the timer. Placement is one of the major parts of the implementation flow that can change the timing behavior of the design, which is why we needed a custom placer for asynchronous logic. Our entire placement flow is implemented in a tool that we call Dali [14].

As a final step, we have implemented a power detailed router. This is a special router used to connect power and ground busses to the gridded cells. The placer reports the mini-row locations, and the mini-row locations are used to route power and ground signals. Finally, power and ground connections to gridded cells are made by the power detailed router from the horizontal metal busses. The power detailed router also supports power grid reinforcement across the entire gridded cell region. Note that this step is only required for gridded cells [15].

2.7 Parallel Global Routing

After placement, the next phase in the automation flow is wire planning, sometimes referred to as global routing. In this step, the layout is divided into a coarse routing grid, and wires are “routed” using only the coarse routing grid. This step also includes information about metal layer assignment for each wire in the design as it passes through different routing grid bins. Figure 6 shows the steps used to develop the coarse routing plan.

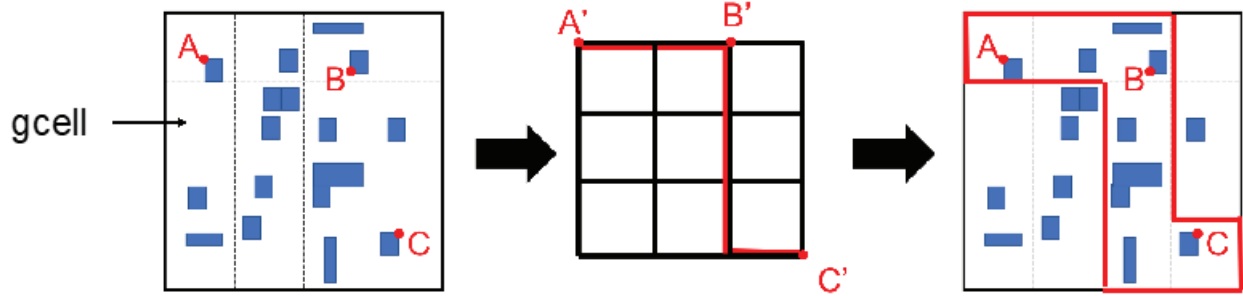


Figure 6. Global Routing on a Coarse Grid

The layout region is divided into “gcells” for coarse wire planning (left). Each gcell is then treated as a point in a routing grid, and horizontal and vertical connections are created between neighboring grid points (middle). Routing is performed on the grid graph, and then the route is converted into a “coarse” route plan in the original layout (right).

To speed up routing, we used the Galois framework to develop a parallel implementation of global routing. Our baseline implementation started with FastRoute 4.1, a high quality open-source academic global router [16]. We include two different forms of parallelism in our implementation. In the early stages of the router where there is plenty of available routing capacity in terms of wiring tracks, our implementation routes multiple nets in parallel. Toward the end when there are capacity constraints and multiple nets must compete for routing resources, we use fine-grained parallel routing per net to explore multiple options for a single net in parallel [17].

2.8 Parallel Asynchronous Static Timing Analysis

Significant advances were made in both the theory and practical realization of static timing analysis for asynchronous logic. In particular, the completed work as part of the IDEA project represents the first static timing analysis engine for asynchronous circuits that can handle non-trivial designs (millions of gates). Our paper on this topic was selected as the best paper at the 2020 IEEE Symposium on Asynchronous Circuits and Systems [18].

Theoretical Advances: We made two key theoretical contributions to the timing analysis of asynchronous logic. These two contributions underpin our implementation of static timing analysis. The first contribution was a result that shows that conjunctively caused (sometimes called AND causal) asynchronous logic exhibits periodic behavior in the general setting. Previous work that is over thirty years old [19], [20] provides a foundation for expecting average periodic behavior for asynchronous circuits. Extensions of that work show that in fact the periodic behavior occurs not just on average, but in fact eventually all signal transitions have periodic behavior [21]. However, these results only apply in the setting when the circuit graph is a single strongly connected component. A simple example where this assumption fails is the bundled data FIFO shown in Figure 2, where the data path combinational logic has no signal path back to the controller. While such a system can be analyzed in an ad-hoc manner by deleting parts of the circuit manually and then analyzing a subset of the system (e.g. just the control logic), such an approach is unusable when developing a general EDA tool for timing analysis.

In a sequence of papers, we showed how the exact periodicity result can be extended in two ways: (i) to handle circuits that are *critically connected*—where there is a path from the critical cycle in the circuit to all other parts of the circuit [22]; and (ii) to handle circuits with multiple critically connected components that interact [23]. Hence, we now have an approach to timing the overall circuit exactly in a way that generalizes—and hence is suitable for implementation in an EDA tool.

Second, we showed that any timing constraints required to order signal transitions (commonly referred to as *relative timing* [24]) can only be guaranteed via a special circuit structure called a *zig-zag pattern* [8]. This zig-zag pattern can be specified in the ACT framework, and our static timing analysis engine can check if the zig-zag pattern components respect the specified timing constraints. This work was selected as the best paper at the 2023 IEEE Symposium on Asynchronous Circuits and Systems.

Practical Realization: All these ideas are implemented in the latest version of Cyclone, a static timing analysis engine implemented in the ACT framework. As part of this effort, our work introduced for the first time the notion of arrival time, required time, and timing slack in the asynchronous circuit setting. The definitions are such that they reduce to their conventional meaning if Cyclone is asked to analyze a clocked circuit.

Cyclone is the first parallel static timing analysis engine for asynchronous circuits. We parallelized Cyclone using the Galois framework (Section 2.3). Note that because we must *determine* the cycle time (unlike clocked circuits where the target cycle time is an input to the timer), we expected that our timer would be slower than one for clocked logic and hence pursued a parallelization strategy from the start of this effort. The net result is that our timing analysis framework has a runtime that is only slightly slower than the runtime for clocked circuits of similar complexity.

Cyclone is also the first incremental timing analysis engine for asynchronous logic. We include numerous optimizations for efficiency so that the time and memory complexity can be controlled by the user of the timing engine. Cyclone has also been integrated into our implementation flow in terms of the placement and global routing steps, but this integration still needs improvement.

2.9 Memory Compiler

Leveraging the OpenRAM python framework, we developed an implementation of a memory compiler for asynchronous logic called AMC (for Asynchronous Memory Compiler). Changes made to OpenRAM to support asynchronous memory macros have been added to a branch of the OpenRAM git repository. AMC includes support for several commercial technology nodes, and we taped out chips in 65nm, 28nm, and 12nm to validate the memory compiler. AMC also includes support for a “read-modify-write” cycle, where data are read from a memory location and then a different value is written back to the same location in a single memory operation. This operation has lower throughput than the traditional read or write operation but is significantly faster than reading followed by writing. The asynchronous nature of our implementation enables us to exploit the timing flexibility afforded by asynchronous logic to support this operation without slowing down ordinary read or write operations [25].

2.10 Industry-standard File Exports

There are a few commercial and open-source tools that are used for the implementation of clocked logic that can be re-used for the implementation of asynchronous circuits. Two concrete examples include circuit simulation (SPICE-level simulation), the detailed routing step in place-and-route, design rule checking, layout versus schematic checking, parasitic extraction, and top-level chip assembly and fill. To enable the IDEA flow to interoperate with third party tools, we developed several different file exports from the ACT design.

SPICE export. Once the user design has been mapped to a gate-level implementation using either production rules or a cell library, the ACT tool `prs2net` can be used to automatically generate a hierarchical SPICE netlist. The tool takes a technology-dependent configuration file that includes information about SPICE model names as well as any specific parameters that must be included in the SPICE output. `prs2net` can also include extra capacitive loads on signals as well as estimated diffusion area and perimeter values so that SPICE simulations provide a bit more realistic results prior to using true parasitic extraction.

The SPICE export functionality is also useful for layout versus schematic checking, and a simplified SPICE netlist can also be automatically generated from ACT for this purpose.

Verilog export. The tool `act2v` can be used to export a hierarchical Verilog netlist for the entire design. This Verilog netlist can be imported into schematic visualization and commercial place and route tools. This feature has been used by some of the users of the ACT flow for analog/mixed-signal asynchronous circuit design.

LEF/DEF export. The place and route flow can export LEF and DEF files at different steps: before any placement; after placement; after power detailed routing; and after global routing. The global routing flow generates the `.guide` format, which has been the standard format used by the academic EDA community.

The final Graphic Design System II (GDSII) file that is required by the foundry can be generated using existing tools from the LEF and DEF output. Examples of tools with this functionality include the `magic` layout editor, which is a widely used open-source chip layout editor.

2.11 Overall Design Automation Flow

The overall software flow is summarized in Figure 7. We developed software to support all the components described, except for those in grey that correspond to third-party tools. The grey components are re-used from any clocked design flow—either other IDEA-developed tools, other open-source options, or commercially available EDA tools.

The components shown in orange use the Galois parallel programming framework to speed up runtime by exploiting multi-core parallel execution.

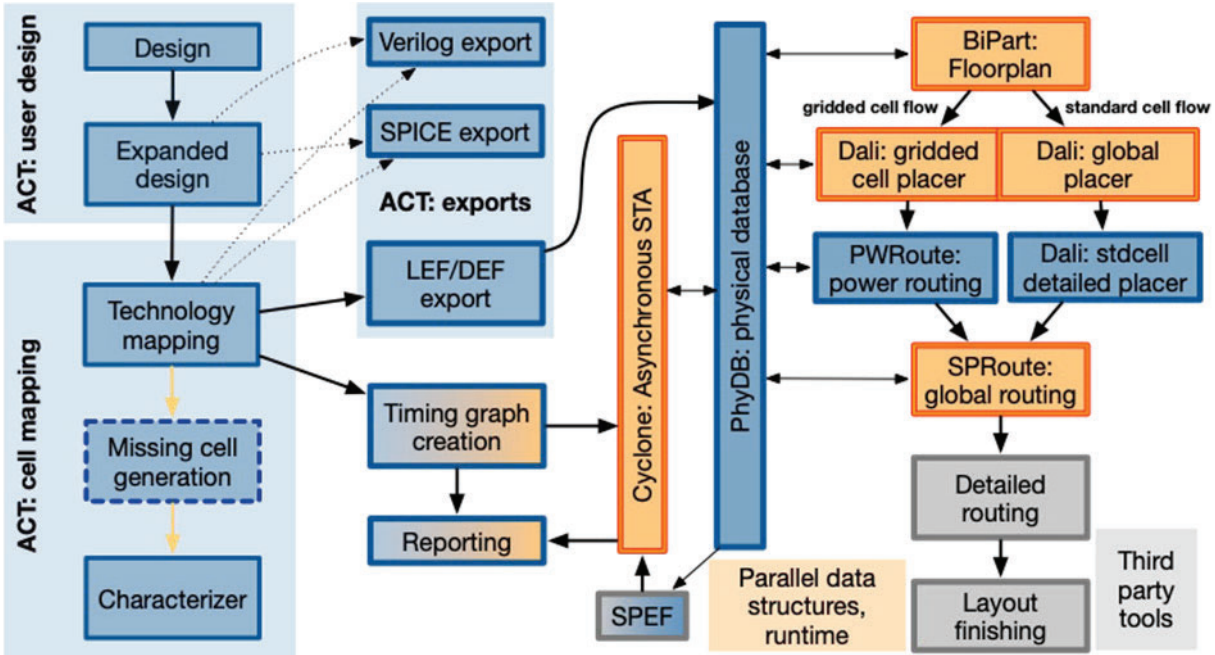


Figure 7. The Overall Implementation Flow Developed by the IDEA Project

The user design is expanded into its gate-level implementation (top left). The implementation is then mapped to a cell library via the technology mapping step. If new cells are required, then we provide support for the user to create the missing cell and then characterize it, and incorporate it into the cell library. The mapped design can be analyzed for timing and is then mapped to LEF/DEF for placement and routing. LEF and DEF representations are held in an in-memory database called phyDB. The place and route tools read and write to this database, and also have an interface to static timing analysis through a unified application programming interface. Our flow stops after global routing, and can hand-off the results to third party tools to finish the routing and complete the design of the chip.

Finally, we have integrated all the tools into a single, unified software package dubbed “interact”—for Interactive ACT implementation flow [26]. Almost all the tools we have developed can be executed directly as interact scripts, and the scripting language includes a full programming language in case the user would like to customize their implementation in any manner.

3 RESULTS AND DISCUSSION

3.1 Community

To support the asynchronous design community, we developed significant amounts of documentation for the tools and flows, including example designs and scripts. The documentation includes details of the ACT language, as well as common use cases and design patterns. We also developed the ACT standard library that contains numerous pre-defined components that are commonly found in chip designs.

Our public documentation is available on a wiki at <https://avlsi.csl.yale.edu/act/>, which is hosted on the Amazon Elastic Compute Cloud. We also have a few third-party contributors to the documentation, including an academic group from Europe and a commercial group using the ACT tools for mixed analog/asynchronous design.

To educate users on asynchronous design, we organized a summer school on asynchronous design in 2022. This school was virtual, and we had over 500 registrants; of those over 300 attended the summer school. Figure 8 shows the distribution of attendees at the summer school in 2022, showing that we reached a broad audience that was divided between academic groups and industry groups.

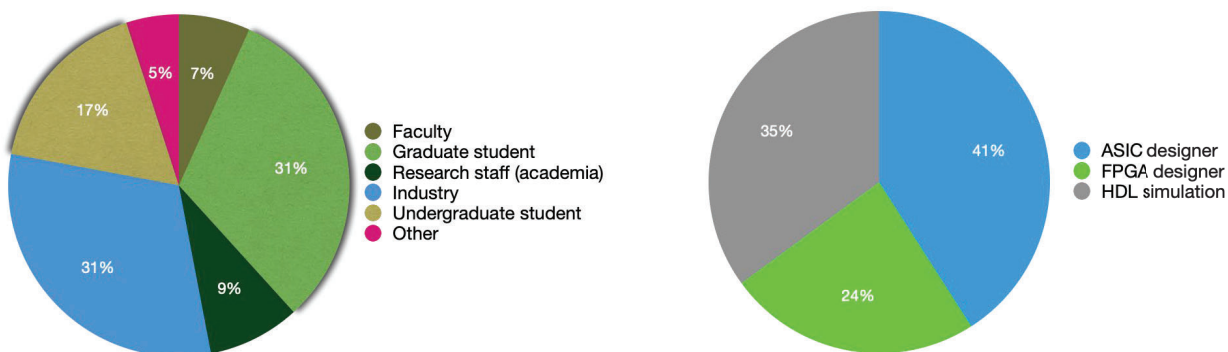


Figure 8. Distribution of Registrations for the Summer School on Asynchronous Design

The school was organized with the support of the steering committee of the IEEE International Symposium on Asynchronous Circuits. Lectures were provided by both the PI, as well as Dr. Ivan Sutherland (Portland State), Dr. Marly Roncken (Portland State), Dr. Benjamin Hill (Intel), and Dr. Montek Singh (University of North Carolina at Chapel Hill). Videos of the summer school are available at <https://avlsi.csl.yale.edu/act/doku.php?id=summer2022:start>

The summer school was divided over three weeks, and offered at a time that would permit industrial participants from the West Coast of the US to participate. Graduate students volunteered their time to provide office hours to answer questions from participants. A number of servers were configured on the Amazon Elastic Compute Cloud for remote login access for any participants that wanted to use our EDA tools without having to install them on their own machines.

The steering committee for the IEEE conference decided last month to repeat this summer school in 2024, also incorporating material from the University of Manchester (UK) and integrating our ACT tools with the Workcraft tools developed in the UK. This will unify all major asynchronous design tools that are publicly available.

We also host an open Mattermost site (an open-source clone of Slack) where users can self-signup for to obtain help on using the ACT tools. The site currently has over 200 registered users.

3.2 Chip Designs

To evaluate the tools in a realistic setting, we taped out several asynchronous chips using the ACT flow:

- We taped out two memory chips in a 65nm process technology and verified the functionality of the memory generated by the memory compiler. We modified some of the circuits generated to improve the performance of the memory.
- We taped out a memory chip in a 12nm FinFET process to validate the memory compiler in an advanced process node.
- We taped out two versions of a microprocessor designed in the ACT framework that implements an instruction set architecture corresponding to a stack machine.

Apart from an issue with the first-generation memory test chip that required an update to the memory compiler, the other chips were tested and found to be functional.

3.3 Intel Study

As part of this effort, Intel evaluated some of the tools developed by the IDEAL flow and compared it to their in-house flow that re-purposing commercial tools for placement and routing. Due to a variety of administrative delays, Intel's work could not start until almost the original end-date of Phase 2 of this project. Hence, the IDEAL-developed tools could not be updated based on Intel feedback as part of this project. However, we the study results have prioritized the features that need to be incorporated into the IDEAL tools for them to be competitive with commercial place-and-route flows; what follows is directly taken from Intel's report.

Intel evaluated Yale's cell placement engine called Dali. We measured quality of results for a set of designs placed using Dali compared against the baseline of Intel Labs' existing flow. For these benchmarks, Dali placements had slightly worse half-perimeter wire length (HPWL) than the existing flow. It is possible to improve Dali's HPWL by increasing local density, but only at the cost of routing congestion. Compared to the existing flow, Dali required 36-51x less peak memory and had 78-136x faster runtime. The primary factor limiting Dali's performance was two-fold:

1. Timing-driven placement is not yet fully functional, and parasitic back-annotation during place and route was not integrated at the time of this evaluation.
2. The IDEAL flow cannot yet modify the gate-level netlist to meet timing during place and route, so designs placed by Dali are not currently suitable for fabrication.

Due to these two limitations, the Yale tools are much less capable than their commercial equivalents, lacking important features required to deliver finished silicon in an advanced process technology. On the other hand, Yale's tools enjoy a huge performance advantage, combined with the fact that they are open-source and built specifically for asynchronous designs. While a daunting amount of engineering effort remains to bring them to feature parity with commercial offerings, they represent a promising foundation for a new generation of design tools.

3.3.1 Benchmarks

To compare placement results for Yale's open-source flow against the Intel Labs commercial baseline, we require a suite of benchmark circuit designs.

Ideally these designs should a) be realistically complex rather than just toy examples, and b) display a range of real-world circuit characteristics (different logic/state ratios, structured vs unstructured floorplans, etc.).

The chosen collection of bundled-data asynchronous designs span two orders of magnitude in complexity, from approximately 1000 movable cells and nets to over 100,000. Some were created specifically for this report, while others are actual designs used in previous research chips.

Each benchmark is composed of different types of cells:

Table 1: Layout Cell Categories

Category	Description	Examples
 Logic cell	<i>basic combinational logic</i>	<i>AND, OR, ADD, etc.</i>
 State element	<i>state holding gates</i>	<i>LATCH, FLIP-FLOP</i>
 BD controller	<i>bundled-data controllers and associated cells</i>	<i>CTRL, DELAY</i>
 Hard macro	<i>large prebuilt cells</i>	<i>SRAM bank</i>
 Physical only	<i>cells with no logic function</i>	<i>well/substate tap, spacer, halo</i>

Hard macros and physical-only cells are typically placed in fixed locations as part of floor planning. This means they are not part of the placement problem, and they serve only as blockages restricting the sites available for placing the other categories of cells.

For generated benchmarks with automatic floorplans, we synthesized the baseline commercial designs using varying global density to provide a greater diversity of design points for comparing the two flows.

3.3.1.1 Multiply-accumulate

The integer multiply-accumulate (MAC) process computes $A = A + B \cdot C$, with parameterized bit-widths for all values. Based on a control signal input, it can optionally clear the internal accumulator state or output the accumulated value to an output channel.

This benchmark is interesting because:

While the MAC is a relatively small and simple process, it contains most of the fundamental constructs used in any asynchronous design: arithmetic computation, local state (accumulator is an algorithmic loop which stores a value across iterations), and conditional communication.

```
int(WA) a;
*[ int(2) ctrl; int(W) b, c;
  CTRL?ctrl;
  [ ctrl==0 -> a=0 // 0: Clear accumulator
  [] ctrl==1 -> A!a // 1: Output current accumulator value
  [] ctrl==2 -> B?b, C?c; a=b*c+a // 2: Read inputs, update accumulator
  [] else -> B?b, C?c; a=b*c+a; A!a // 3: Read, update, output result
  ]
]
```

Figure 9: Multiply-accumulate CSP Circuit Description

For the benchmark example used in this report, data width $W=16$ bits and accumulator width $WA=18$ bits.

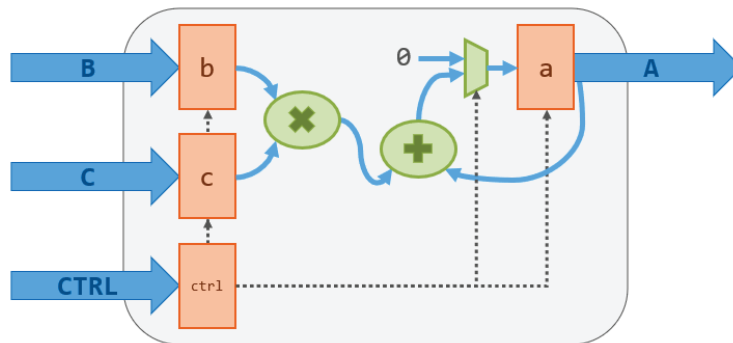
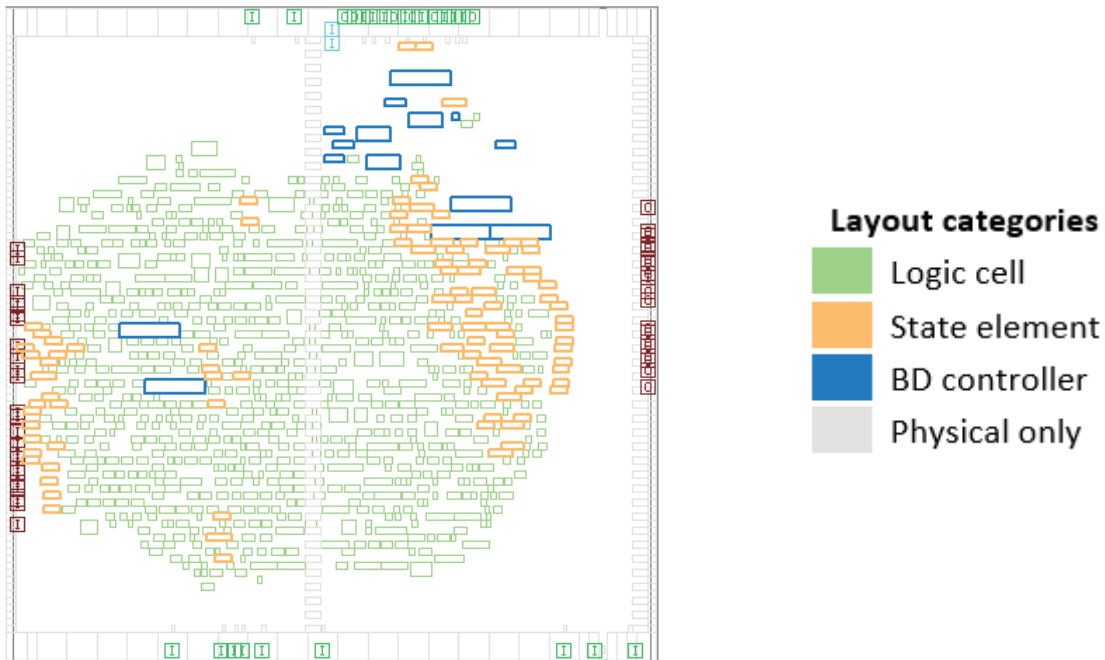


Figure 10: Multiply-accumulate Dataflow Diagram

Table 2: Multiply-accumulate [25%] - basic composition

Cell type	Count		Area
	IOs	68	-
	Nets	999	-
	Movable	881	19%
	Fixed	352	14%
	Sequential	102	3%
	Buffer/inverter	108	1%
	Utilization	-	22%

**Figure 11: Multiply-accumulate [25%] Physical Placement Example (commercial flow)**

3.3.1.2 Floating-point multiply-add

The floating-point fused multiply-add (FMA) process computes IEEE 754 compliant (IEEE Computer Society, 2008) arithmetic, with parameterized bit widths. For the benchmark example used in this report, data width $W=32$ bits. Depending on the value of CMD input, the FMA process can perform multiplication, addition, subtraction, and fused-multiply-add, as well as related operations such as comparison, negation/absolute value, and min/max. The FMA function can have internal pipelining to increase throughput. The benchmark example used in this report has a datapath pipeline depth of 6 stages.

This benchmark is interesting because:

The FMA process is substantially larger and has more complex arithmetic than the MAC (10x the number of standard cells), but its basic process structure is simpler: a predominantly feed-forward pipeline with computation. FMA features internal pipeline stages, so it has a higher percentage of sequential cells.

```
// (... local variable declarations omitted)
*[ CMD?cmd, A?a, B?b, C?c;           // Read inputs and desired operation
  RES!fma_function(cmd,a,b,c,f);      // Compute and send result
  FLAGS!f                             // Send flags (e.g. under/overflow)
]
```

Figure 12: Floating-point Multiply-add CSP Circuit Description

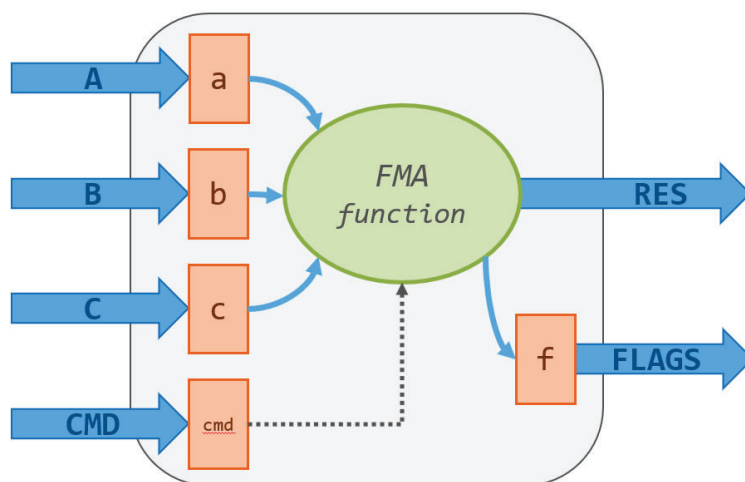


Figure 13: Floating-point Multiply-add Dataflow Diagram

Table 3: Floating-point Multiply-add [60%] - basic Composition

Cell type	Count		Area
	IOs	158	-
	Nets	8700	-
	Movable	8297	53%
	Fixed	596	9%
	Sequential	1971	20%
	Buffer/inverter	736	2%
	Utilization	-	58%

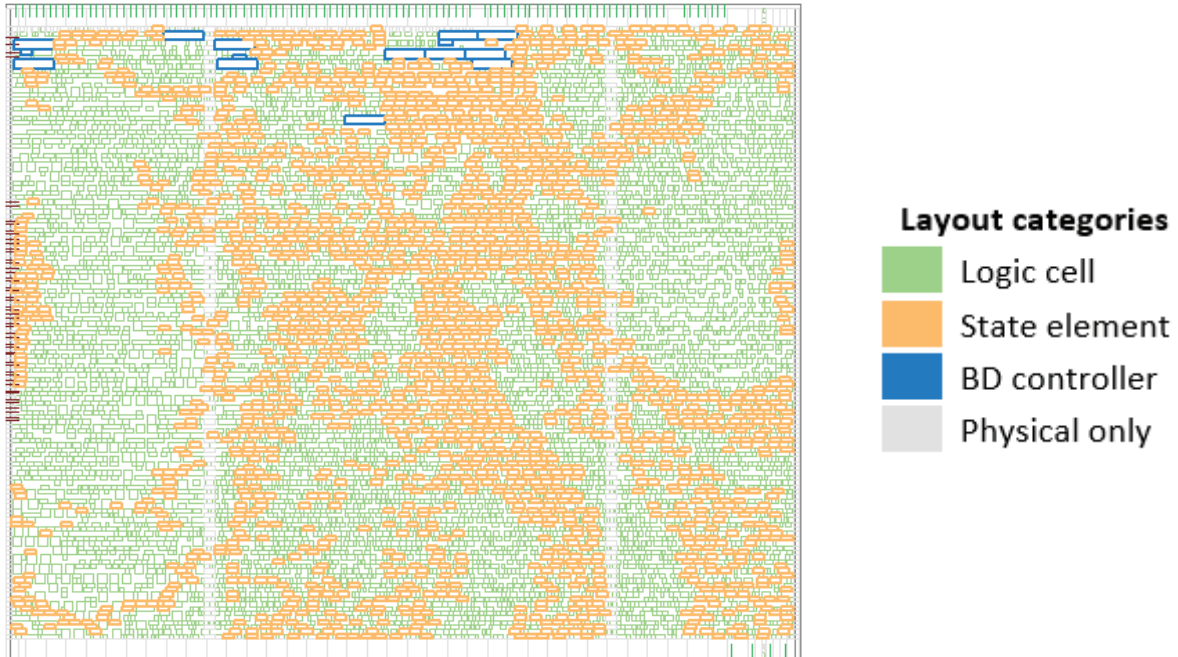


Figure 14: Floating-point Multiply-add [60%] Physical Placement Example (commercial flow)

3.3.1.3 Processor

RISC-V RV32IM processor core, implementing the RISC-V 32-bit base integer instruction set with extension for integer multiplication/division.

This benchmark is interesting because:

A microprocessor is a large and complex design, including computation and multiple control loops. The floorplan for this design was manually created by human experts, and it includes significant pre-placed hard macros for memory as well as the register file.

Table 4: Processor - basic Composition

		Count	Area
Cell type	IOs	165	-
	Nets	37087	-
	Movable	29922	4%
	Fixed	27840	82%
	Sequential	11240	81%
	Buffer/inverter	2978	0%
	Hard macro	16	79%
	Utilization	-	24%

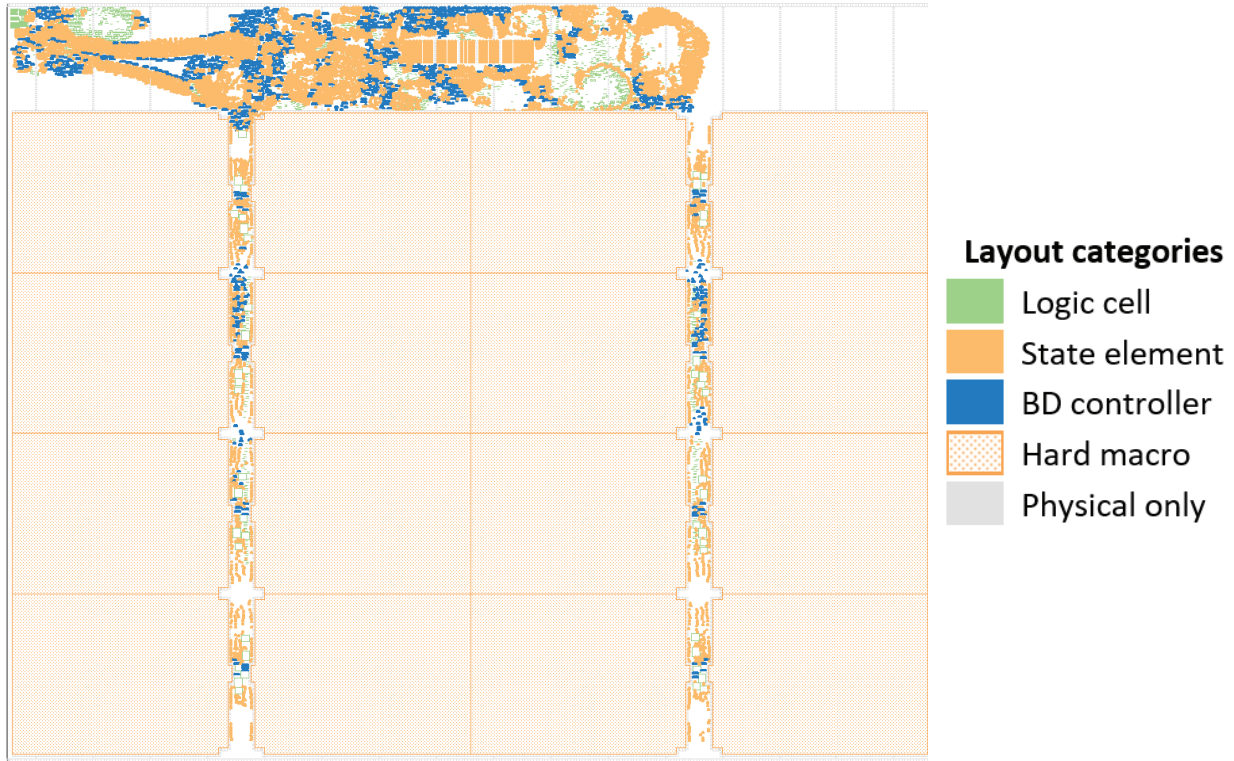


Figure 15: Processor Physical Placement Example (commercial flow)

3.3.1.4 Complex neuromorphic designs

These designs are taken from actual neuromorphic spiking neural network chips. As a result, each one has a single manual floorplan and corresponding global density/utilization.

3.3.1.5 Neuromorphic Core A

This benchmark is interesting because:

The partition has a large number of movable cells and nets for the placer to contend with (double that of the processor). It also has a skewed rectangular aspect ratio compared to the nearly square floorplans of previous designs, which can complicate the placement problem since poor placements may have high wirelength.

Table 5: Neuromorphic Core A - basic Composition

		Count	Area
Cell type	IOs	2015	-
	Nets	74062	-
	Movable	68136	39%
	Fixed	7758	10%
	Sequential	23721	22%
	Buffer/inverter	7560	2%
	Utilization	-	44%

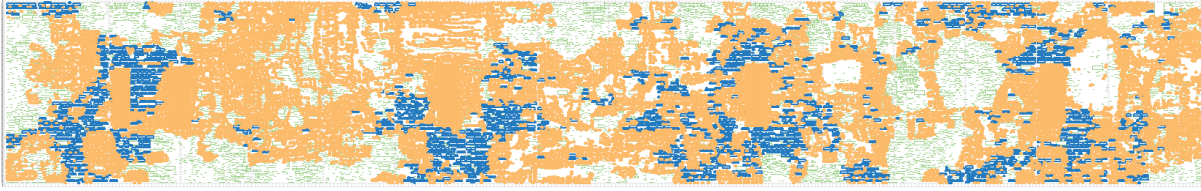


Figure 16: Neuromorphic Core A Physical Placement Example (commercial flow)

3.3.1.5.1 Neuromorphic Core B

This benchmark is interesting because:

It has the largest cell count of any design considered in the report, and as discussed in the compute resource usage section (page 42) it is close to the upper limit of reasonable runtime for a single partition using the commercial placement flow. This runtime limit is one of the main reasons the single logical Neuromorphic Core has been divided into smaller physical partitions in the first place. If the Yale flow can improve the performance they could be run as a single partition, greatly saving design effort and complexity. This design also features a non-rectangular floorplan, with a notch in the upper left corner to accommodate another physical partition.

Table 6: Neuromorphic Core B - basic Composition

Cell type	IOs	1835
	Nets	126723
	Movable	119200
	Fixed	18372
	Utilization	47%

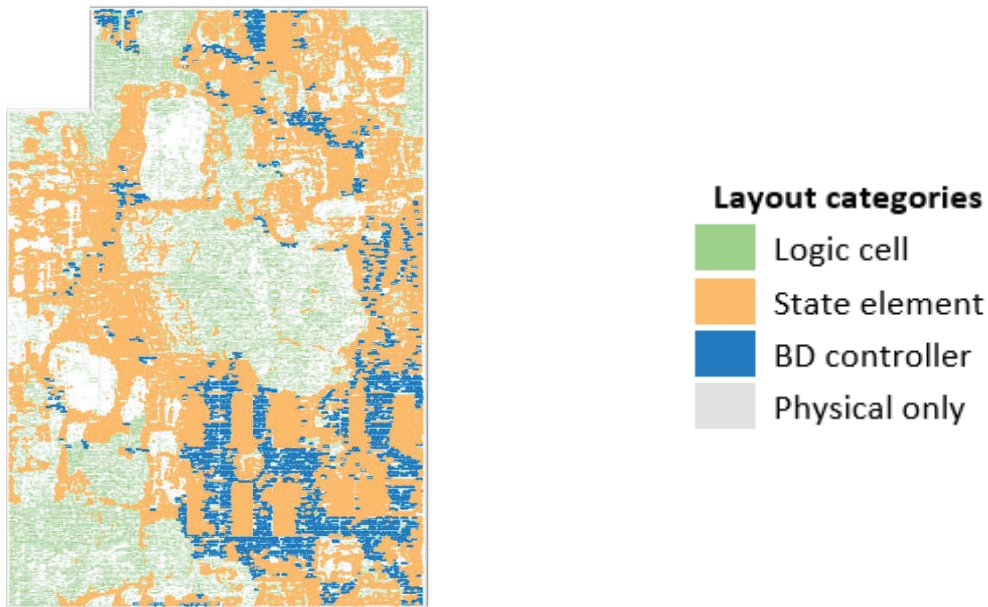


Figure 17: Neuromorphic Core B Physical Placement Example (commercial flow)

3.3.1.6 Network-on-chip router

This benchmark is interesting because:

Due to the large number of channels that need to traverse the core of the router, it has a highly structured floorplan with most cells being manually pre-placed. This design also uses placement regions and blockages, discussed further on page 40.

Table 7: Router - basic Composition

Cell type	IOs	1291
	Nets	21654
	Movable	8537
	Fixed	16033
	Utilization	24%

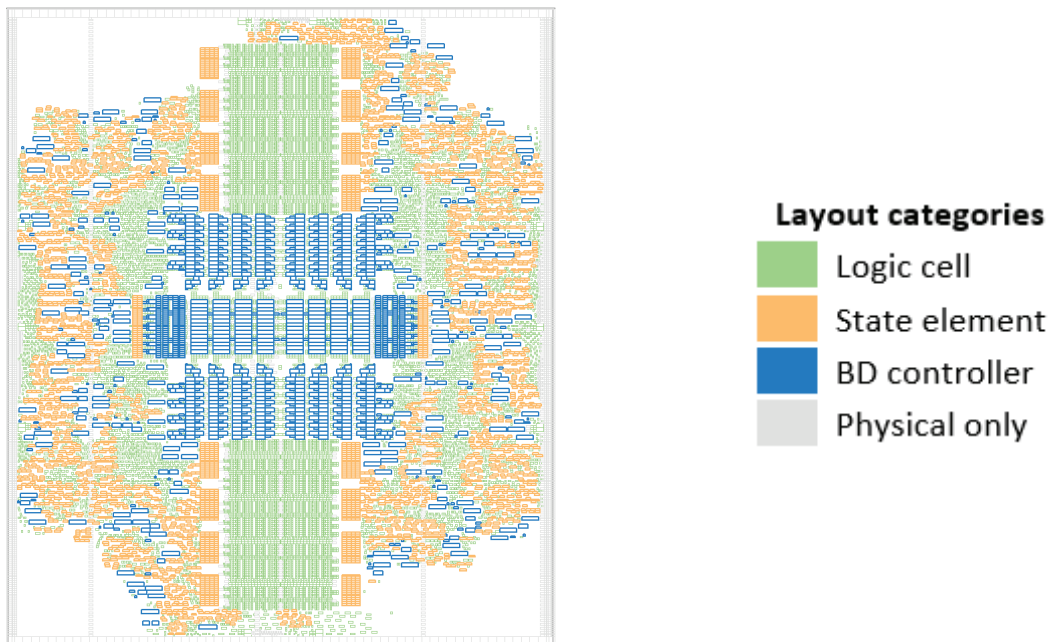


Figure 18: Router Physical Placement Example (commercial flow)

3.3.2 Placement

In this section we report experimental results comparing cell placements generated by the Yale flow to placements from the baseline Intel Labs flow, both targeting an advanced Intel process technology. Results are reported comparatively, as the percent improvement or degradation for the Yale flow.

In this report we will consider the full set of quality of results (QoR) metrics, not just those that Dali is currently capable of directly optimizing (i.e. half-perimeter wire length).

Table 8: Key Quality of Results Figures of Merit

Metric	Definition
Half-perimeter wire length (HPWL)	For a given routing net, construct a bounding box around all terminals of the net. Width plus height of this bounding box is the HPWL for the net. In the simplest case of a two-terminal net, this is the Manhattan distance for a wire connecting the two terminals. HPWL is used as an easy to compute proxy for wiring cost, and cell placement tools often attempt to minimize total HPWL for a design. Lower is better: in general, shorter wire lengths lead to decreased delay, energy, and routing congestion.
Timing slack	For a given signal path and timing constraint, the difference between required signal arrival time and actual arrival time. A negative value indicates that the constraint has been violated, i.e. timing is not met. Useful slack-related metrics include: • worst-case negative slack: sets the cycle time/clock speed for a design • number of violating paths, total negative slack: indicate whether timing issues are localized or widespread
Congestion	A design region is said to be congested when multiple signal nets are competing for limited wiring resources. This makes routing those nets more difficult and in some cases impossible, which requires re-placing the design. In the placement phase before detailed routing, a design is often divided into coarse global routing cells. For each of these regions, we can estimate the total number of signals passing through the region. Overflow occurs when the number of signals is higher than the number of available routing tracks within the region.
Density/utilization	Ratio of cell area to total silicon area. Several variants can be computed, for example including/excluding physical-only or fixed cells. All things being equal, higher utilization is better: less silicon area consumed = lower cost, shorter wiring distance between cells often means higher performance and lower energy. Unfortunately, high placement density can also lead to congestion and routing failures, which sets a practical limit on maximum utilization for a given design.

3.3.2.1 Evaluation Method

To fairly compare the two flows, we need to evaluate them on the same placement problem. Specifically, they should place the same netlist within the same floorplan using the same constraints.

Yale’s placer Dali is a straightforward placement engine, taking an input netlist and placing each of its components in a legal location. By contrast, the Intel flow fuses several of the stages of the EDA process. During placement, the Intel flow modifies the design netlist including re-sizing cells based on their physical proximity to the successor cells whose inputs they drive. These alterations to the netlist make it more difficult to fairly compare the flows.

Ideally, we would provide the same post-synthesis netlist and floorplan to each flow and allow each to modify the netlist to meet timing and other constraints based on physical placement decisions. Because this is not currently a capability of the Yale flow, we use the following procedure in this report:

Generate initial design netlist

Start with the placed netlist generated by Intel’s flow in an advanced process technology – this is the commercial baseline result. During placement the commercial placer may have optimized the original post-synthesis netlist.

Un-place design

Edit the commercial DEF to remove PLACED status for all standard cells in the design. Retain all floorplan information from initial design: pin placement, location for FIXED physical-only cells, hard macros, etc.

Re-place design

Provide the un-placed design netlist and floorplan as input placement problem to Dali. Sweep Dali target density and compare QoR to commercial results.

Global density (total cell area of the design as a percentage of available floorplan area) has a strong impact on both QoR and runtime for the cell placement task. In general, denser designs can have better QoR assuming routing congestion can be managed, but the placement runtime will be longer because there is less free space within the floorplan for the placer to use.

Floorplans for the MAC and FMA benchmarks were automatically generated. For these designs we ran separate placement trials at different utilization ratios. Note that the starting netlists for these trials will be similar but not identical, since the synthesis tool may choose e.g. a different adder topology to trade area for performance. Where applicable, global density is reported along with the benchmark (rounded to the nearest 5%). For example, “FMA [60%]” represents the Floating-point multiply-add benchmark synthesized at 60% utilization ratio.

For all other designs the floorplan was manually created by human experts. These benchmarks will have only a single global density. In all cases the same floorplan and netlist was used for both Dali and the commercial flow.

In addition to design QoR, we also collected compute resource usage metrics (peak memory usage and total runtime) for each placement trial.

3.3.2.1.1 Half-perimeter Wire Length

In one sense this is the most relevant metric for this phase of the project, since it is the only one that Dali currently optimizes. Commercial tools consider a broader set of metrics (e.g. timing, routing congestion) when performing cell placement, so they may appear less competitive in this category while still delivering superior QoR overall on the metrics that matter in silicon (performance, energy, area).

For each benchmark, we performed the placement trials described in the previous section: one with the commercial flow as baseline, and several Dali trials with different target densities (which must be strictly greater than the actual benchmark's global density). We then compute HPWL for each completed placement, in the X and Y routing directions separately and as a combined total wirelength.

Figure 19 shows an example HPWL comparison for the FMA [60%] benchmark. The commercial baseline is shown as a dashed line (1.0), with the global density indicated with a star. Each Dali trial at a given target density is plotted as the ratio $HPWL_{Dali}/HPWL_{commercial}$. Values below 1 indicate that the Dali placement had lower (better) HPWL, while values greater than 1 mean the commercial HPWL was lower.

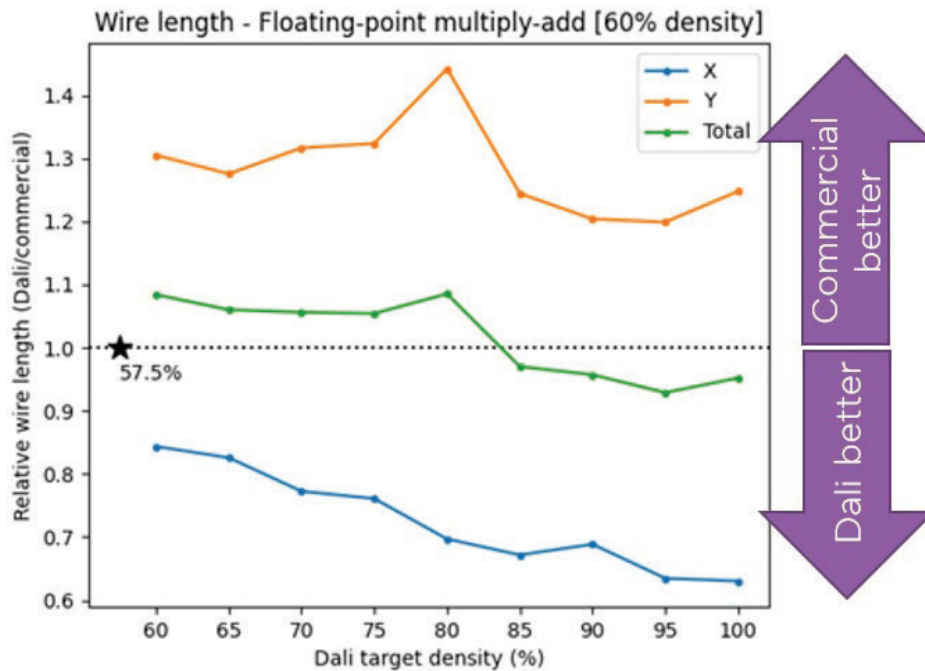


Figure 19: Half-perimeter Wire Length Comparison for Floating-point Multiply-add [60%]

Some observations from this specific example that hold true for most or all benchmarks:

- Dali performs worse on total HPWL at the lowest target density, which is perhaps the most direct comparison between the two flows. However, the difference is small (around 10%)

for this and most benchmarks). Furthermore, Dali runs significantly faster than the commercial flow (see compute resource usage on page 42). If we were to increase the effort Dali spends finding a placement solution, it is likely that this small HPWL performance gap would disappear.

- As Dali target density increases HPWL improves, usually beating the commercial baseline. This is encouraging, but it does not necessarily mean that Dali is the better placement tool. Recall that in these trials we hold the floorplan constant, so a more direct comparison might be to rerun the commercial flow with a smaller floorplan/higher utilization to match Dali's target density.
- Compared to the commercial placer, Dali reduces HPWL better in the X routing direction than in Y. This may be an artifact of Dali's roots as a gridded cell placer. In any case, it could likely be adjusted via a tuning parameter if necessary.

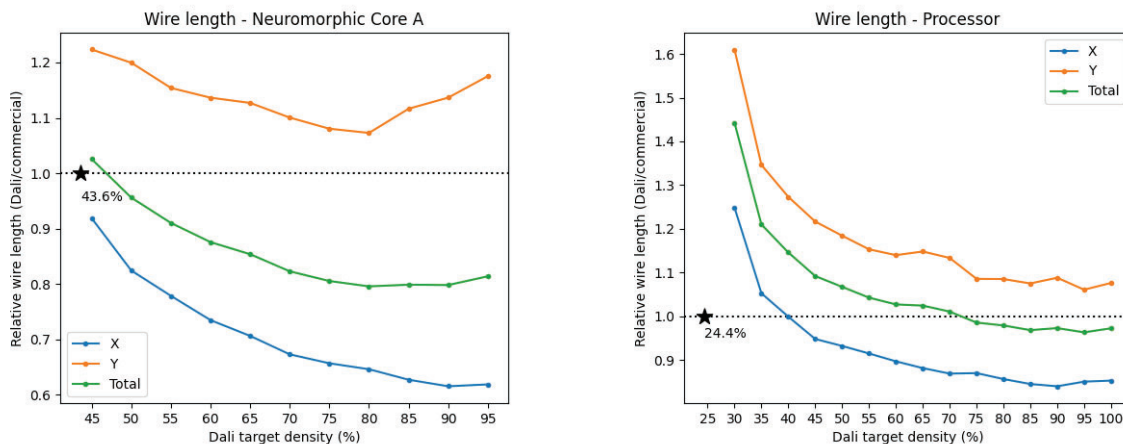


Figure 20: Half-perimeter Wire Length Comparison for Neuromorphic Core A and Processor

Focusing on two of the most complex benchmarks, Figure 20 highlights one example where Dali shines and another where it struggles.

Neuromorphic Core A (left) is much wider than it is tall (see Figure 16), so Dali's superior performance in the X direction contributes heavily to its superior total wirelength performance. In the next section, we will see in more detail how increasing local density changes Dali's placement results.

The floorplan for the Processor (right) benchmark is dominated by fixed hard macros (see Figure 15) with relatively narrow channels between them to place the rest of the cells. At low target densities, Dali is unable to effectively use the fragmented remaining area, leading to some of the

worst HPWL of any benchmark (more than 40% worse than the commercial baseline). At higher target density, Dali can pack the moveable cells together, achieving HPWL parity with the baseline around 70% target density.

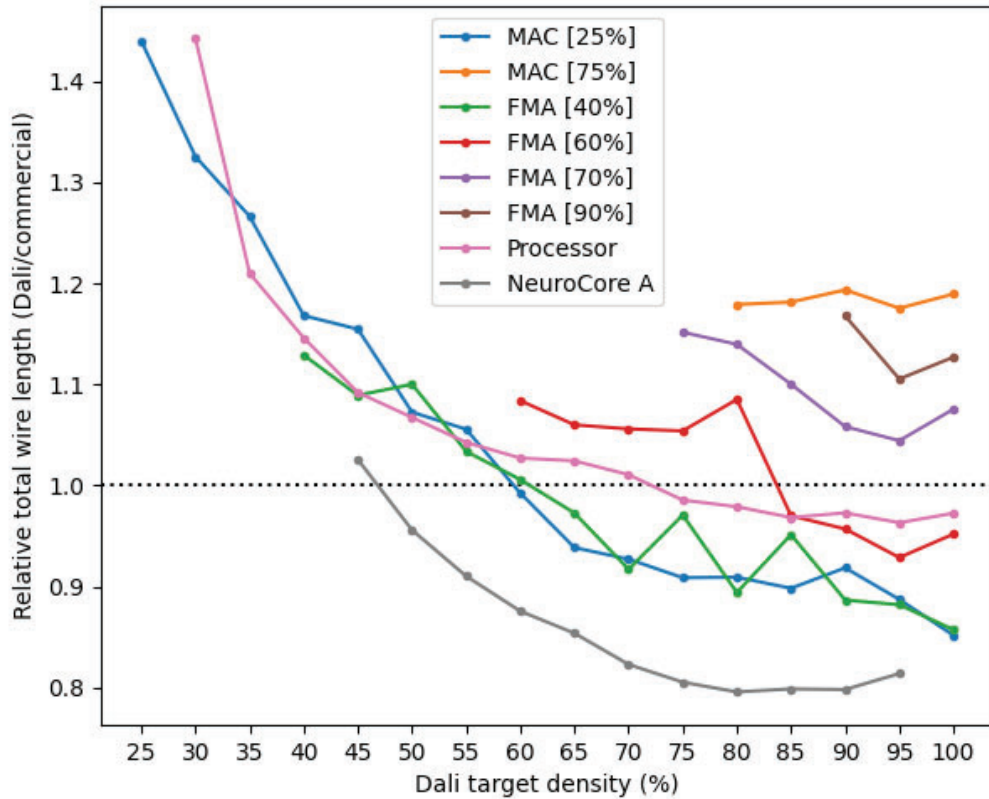


Figure 21: Half-perimeter Wire Length Comparison for All Designs

Figure 21 shows the relative HPWL for placements of all benchmarks by each of the two flows. For the most part, all benchmarks follow the same trend as discussed above.

Dali has a difficult time reducing HPWL for low utilization designs at low target density. In addition to the Processor benchmark discussed above, this is evident for MAC [25%]. In the next section we will more closely investigate differences between the Dali and commercial placements that account for this result.

There is a cluster of designs in the upper right corner (MAC [75%], FMA [70%], FMA [90%]) for which Dali does not match the commercial placer HPWL at any target density. The common factor for these designs is high utilization for the baseline design. This suggests that it is Dali's ability to pack designs into a smaller footprint within a given floorplan by increasing target density that accounts for its improved HPWL performance. When the target density is close to the actual design utilization, the commercial placer has the edge in HPWL as noted above.

3.3.2.1.2 Density

The main Dali tuning parameter we explored in the previous section was target placement density. In this section, we will study in detail how changing this parameter impacts the cell placement quality.

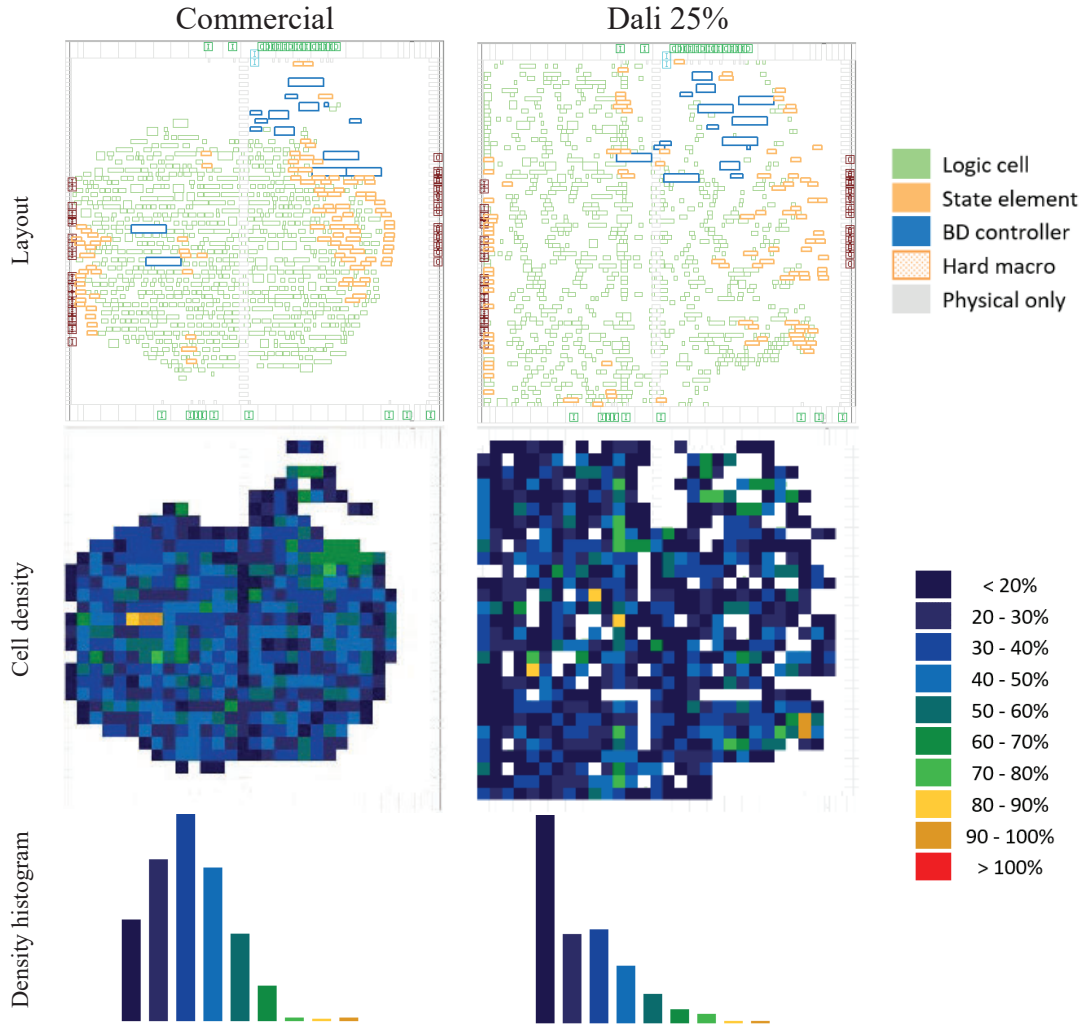


Figure 22: Multiply-accumulate [25%] – commercial vs Dali Placement

Figure 22 shows the placement results for the commercial flow (left) and Dali (right) for the MAC [25%] benchmark in the case where Dali's target density is close to the global utilization. The top row shows where each type of cell has been placed; the middle row is a heat map representing the local placement density on a coarse grid; the bottom row is a histogram of local placement densities for each grid bin.

Comparing these placements, the reason for the difference in HPWL shown in Figure 21 is immediately evident. The commercial placement clusters the cells into a compact blob, in which most grid bins show local density between 30-40% but there are some higher and some lower including many empty bins so that the global density is around 25%. By contrast, the Dali

placement spreads out the cells more evenly across the entire floorplan, resulting in more bins with low local density. Because the cells are more spread out rather than clustered as in the commercial placement, the average distance between them increases as does HPWL.

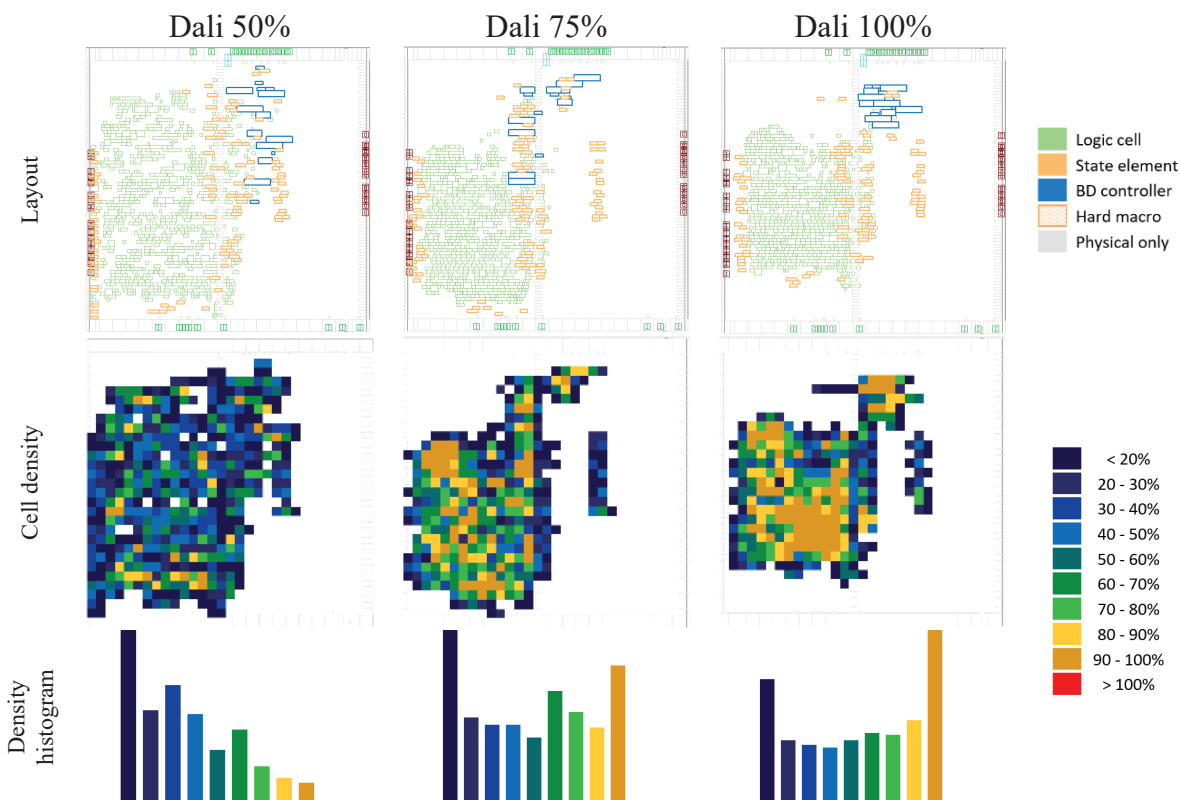


Figure 23: Multiply-accumulate [25%] – varying Dali Target Density

We can increase the Dali target density beyond the global utilization value as shown in Figure 23, which goes all the way to an (unrealistic) 100% target density. As the target density increases, Dali generates more compact placements, so that within the cluster of cells most grid bins have local density near the target. Outside of the tightly packed region, the rest of the floorplan is left empty.

This tight clustering explains how HPWL decreases with increased target density. As with the commercial placement, grouping cells together reduces the average distance for any route between them. There may still be long routes to the I/Os of the partition, which remain at their fixed location in the floorplan, but since there are relatively few of these connections compared to the number of internal nets, they do not change the overall result.

Utilization is quite low for this benchmark, so the clustering effect described is exaggerated, but we see similar behavior in other benchmarks including those with higher utilization. As the target density increases, Dali places most cells in a tighter cluster and leaves the remainder of the floorplan unoccupied, which improves HPWL. If the global density is low enough, Dali can compress the cells enough to surpass the HPWL of the commercial placement.

Given the placement results studied so far, it appears that increasing target density is an unalloyed positive: it decreases HPWL and potentially allows for smaller floorplans. This is not always the case, as we will see by reviewing the impact of density on routing congestion in the next section.

3.3.2.1.3 Routing Congestion

The next step in the EDA flow following cell placement is routing. It is critical that placements allow for designs to be successfully routed using the constrained wiring routing resources available in a particular fabrication process technology.

If there are too many routes competing for the same wiring tracks in one area, some of those routes must take longer paths around the congested region. In the worst case, it becomes impossible to complete all netlist routes given the available wiring resources.

At the placement stage, we can evaluate global routing track congestion as a proxy for future routing quality. Instead of considering individual wires as in detailed routing, the floorplan area is divided into coarse global routing cells. Each of these cells has a capacity based on the number of routing tracks within the cell. Overflow occurs when the number of routes passing through a global routing cell is greater than its capacity.

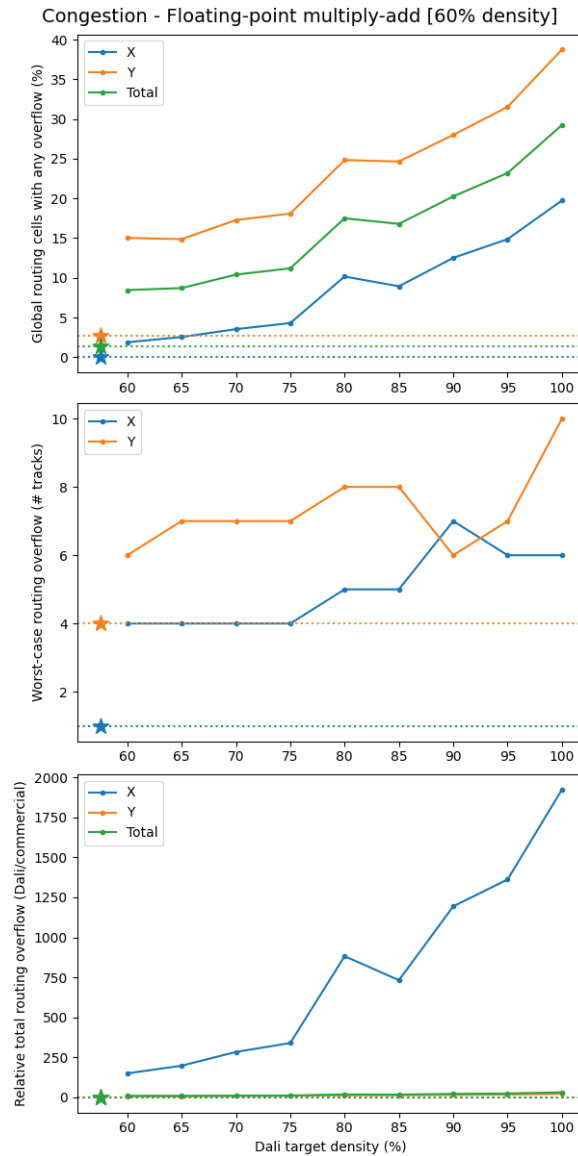


Figure 24: Floating-point Multiply-add [60%] - global Routing Congestion

Figure 24 shows several congestion metrics for the FMA [60%] benchmark. As in the HPWL data, the commercial baseline is shown as a dashed line, with the global density indicated with a star. The solid lines show each Dali trial at a given target density, with X/Y/total results plotted separately where appropriate.

The top graph shows the percentage of global routing cells that have *any* overflow, even if it's only by one track. For this benchmark, the commercial placer is in the low single digits. At low target density, Dali overflows fewer than 10% of global routing cells, but as the density increases the overflows become more widespread, rising above 25%.

The middle graph shows the worst-case overflow across all global routing cells. For this benchmark, the commercial placement has a maximum overflow of 1 and 4 tracks in the X and Y

routing directions respectively. The worst-case overflow for Dali is a bit worse, between 4-7 tracks in X and 6-10 tracks in Y.

These metrics must be considered together to understand the routing congestion severity. If there is overflow in 50% of global routing cells, that may not be a major problem if it is only one track. Conversely, if the worst-case overflow is twenty tracks, but it only occurs in a single global routing cell, it is easier to deal with that localized routing hotspot and the overall placement may still be good.

The bottom graph represents one possible summary metric: it compares the sum of all overflows across all global routing cells for the two flows. At first glance, the overflow in the X routing direction for Dali on this benchmark looks somewhat alarming. In this case, it is more that the commercial placer performed very well (only one total track of X overflow) rather than Dali necessarily performed poorly. Total overflow in both directions is a better metric to consider, and here Dali overflows between 8-30x the number of tracks compared to the commercial baseline.

In addition to these numerical congestion metrics, we can examine the placement layout to see visually where congestion occurs and its relative severity. Figure 25 extends the placement density plots we've previously reviewed with an additional row depicting global routing congestion. Each global routing cell with overflow is represented by a line segment, color coded by the number of overflowing tracks.

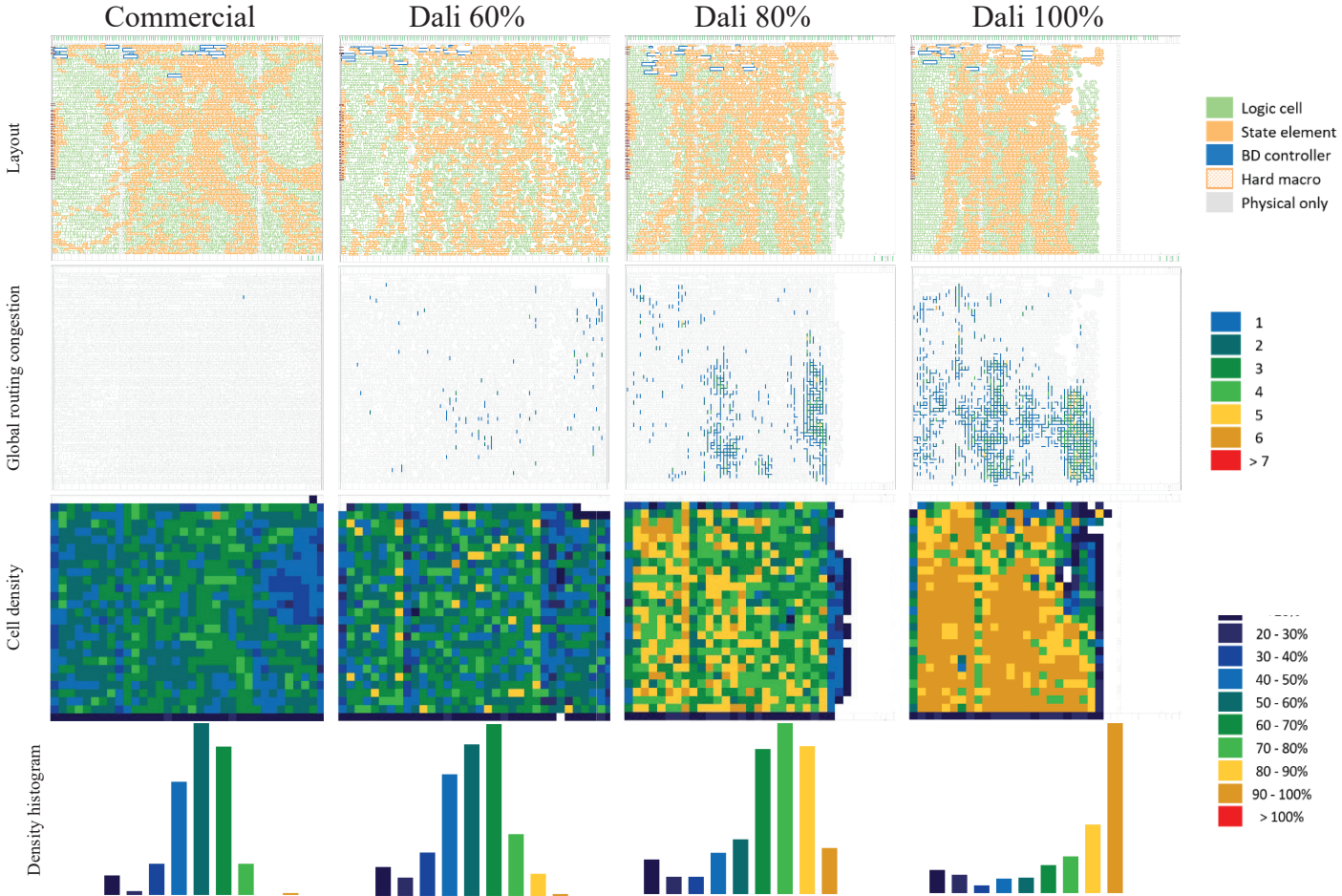


Figure 25: Floating-point Multiply-add [60%] – placement with Congestion

Because the initial utilization for this benchmark was higher, the placement results for the commercial baseline and Dali at lowest target density (60%) are much more similar than they were for the previous MAC [25%] example (Figure 22).

As Dali target density increases the cell placement is compacted, as it was for MAC [25%] (Figure 23). For this benchmark, the placement shifts from a rectangle filling the entire floorplan at 60% target density to a tight square squeezed over to the left side at 100%.

The routing congestions plots in the second row of Figure 25 highlight the drawback of increasing density. The commercial placement has almost no overflow and will be easy to route. Dali at 60% target density has small, scattered areas of congestion: worse than the baseline but quite manageable. At 80% target density, the Dali placement starts to show significant congested regions. Contiguous areas of routing congestion are more difficult to handle than scattered overflows because routes diverted around of the area due to overflow must go farther to reach their destination, and these circuitous routes cause secondary congestion elsewhere. Dali at 100% target density has widespread congestion, including hotspot areas with very high overflow. This placement would be challenging to route successfully.

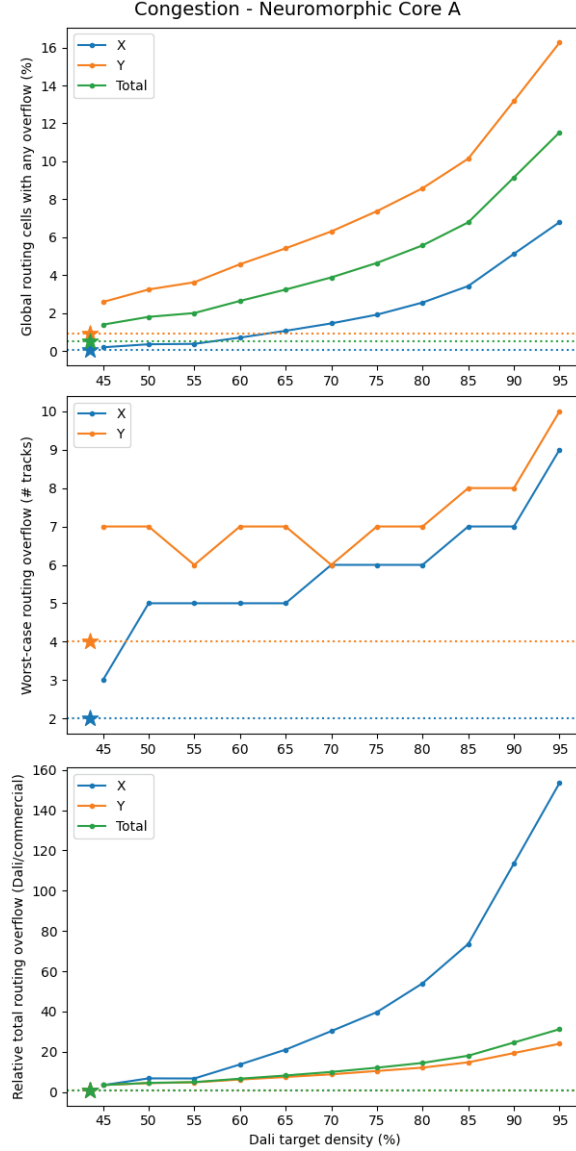


Figure 26: Neuromorphic Core A - global Routing Congestion

Figure 26 shows congestion metrics for the much more complex Neuromorphic Core A benchmark. These results are broadly similar to those for FMA [60%]. The Dali placement trials result in 2-10% of global routing cells with overflow, vs 1% for the commercial benchmark. Worst-case overflow is worse for Dali (3-9 tracks vs 2 in the X routing direction, 6-10 tracks vs 4 in Y), and total overflow compared to the commercial baseline ranges from 3-31x worse depending on Dali target density.

3.3.2.1.4 Timing Slack

The ultimate goal of the placement task is a cell layout that can be routed while meeting all timing constraints for the design. Timing is not directly considered by Dali in these placement results – instead, Dali is currently optimizing HPWL, which is meant to be a proxy for delay.

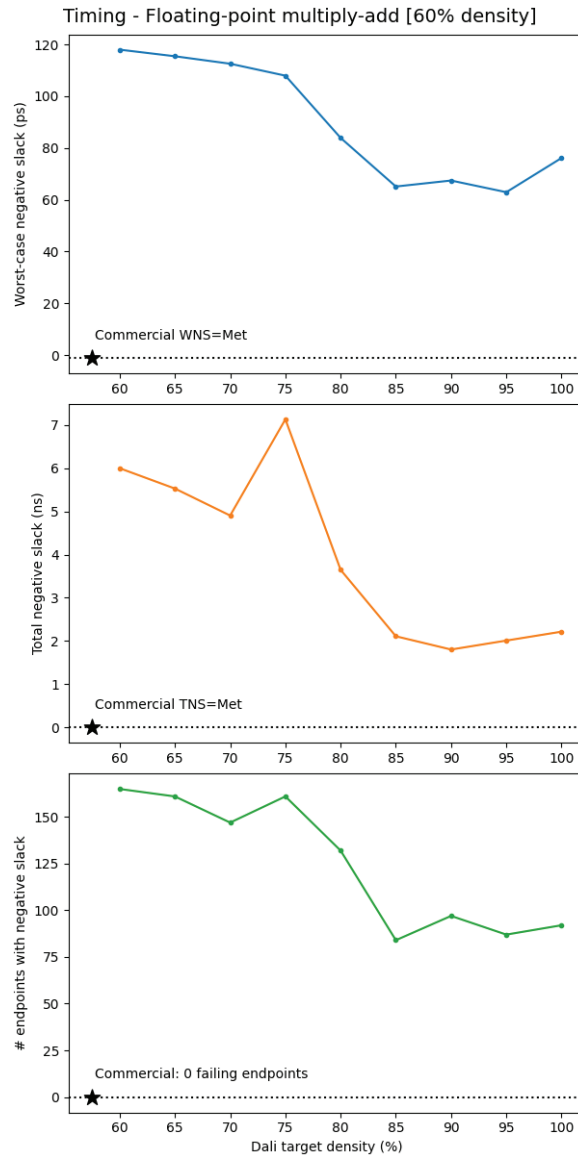


Figure 27: Floating-point Multiply-add [60%] - timing Slack

The commercial tools *do* consider estimated delay during placement. Since timing is ultimately the metric that matters for design correctness and performance, in this section we examine timing results, with the caveat that the comparison will necessarily be unfair to Dali.

Figure 27 revisits the familiar FMA [60%] benchmark, this time examining how well the Dali placement trials succeed at meeting required timing constraints (that again, Dali is not yet directly optimizing for). As in previous plots, the commercial baseline is shown as a dashed line, with the global density indicated with a star. The solid lines show timing negative slack for each Dali trial at a given target density.

The top plot shows the worst-case negative slack (WNS), i.e. the timing constraint that has been violated by the largest margin. In a synchronous design, this would represent the amount by which you must extend your clock period to meet timing. For bundled-data asynchronous

designs the situation is similar, since if all else fails we can usually correct slack violations by increasing delay on handshakes.

For this benchmark, the commercial placer can meet timing (WNS = 0). Depending on target density, Dali WNS is between 60-120ps. This means that the Dali placement trials will have lower performance or will require additional timing engineering change order (ECO) work to meet timing.

The middle plot reports total negative slack (TNS), the sum of margins for all violated timing constraints. The bottom plot shows the total number of timing constraints that are violated. Taken together, these metrics present a picture of the overall timing health of a placement trial. For example, if there are many violating paths but each only misses timing by a small margin, this is a minor problem and TNS will be relatively low. If WNS is high but there are only a small number of failing endpoints, that might indicate a localized problem that can be fixed in the design.

For the FMA [60%] benchmark, Dali TNS is between 2-6ns and there are 75-150 failing endpoints. Given that there are nearly 2000 sequential elements in the design (see Table 3), approximately 5% of them do not meet timing, and on average they do so by a few tens of ps. This is a manageable task for a timing ECO, mostly because FMA is a relatively small benchmark, but the commercial trial result proves that there exists an alternate placement of the same netlist that already meets timing.

In general, the timing slack metrics improve with increasing Dali target density. This tracks with the HPWL results presented previously. It also makes intuitive sense: as the designs become denser, the distance each signal must travel and therefore its latency is shortened. Unfortunately, this simple argument neglects the effects of routing congestion discussed in the previous section. At high densities, congestion will force some routes to take circuitous paths to their destinations, which will ultimately increase latency and worsen timing slack.

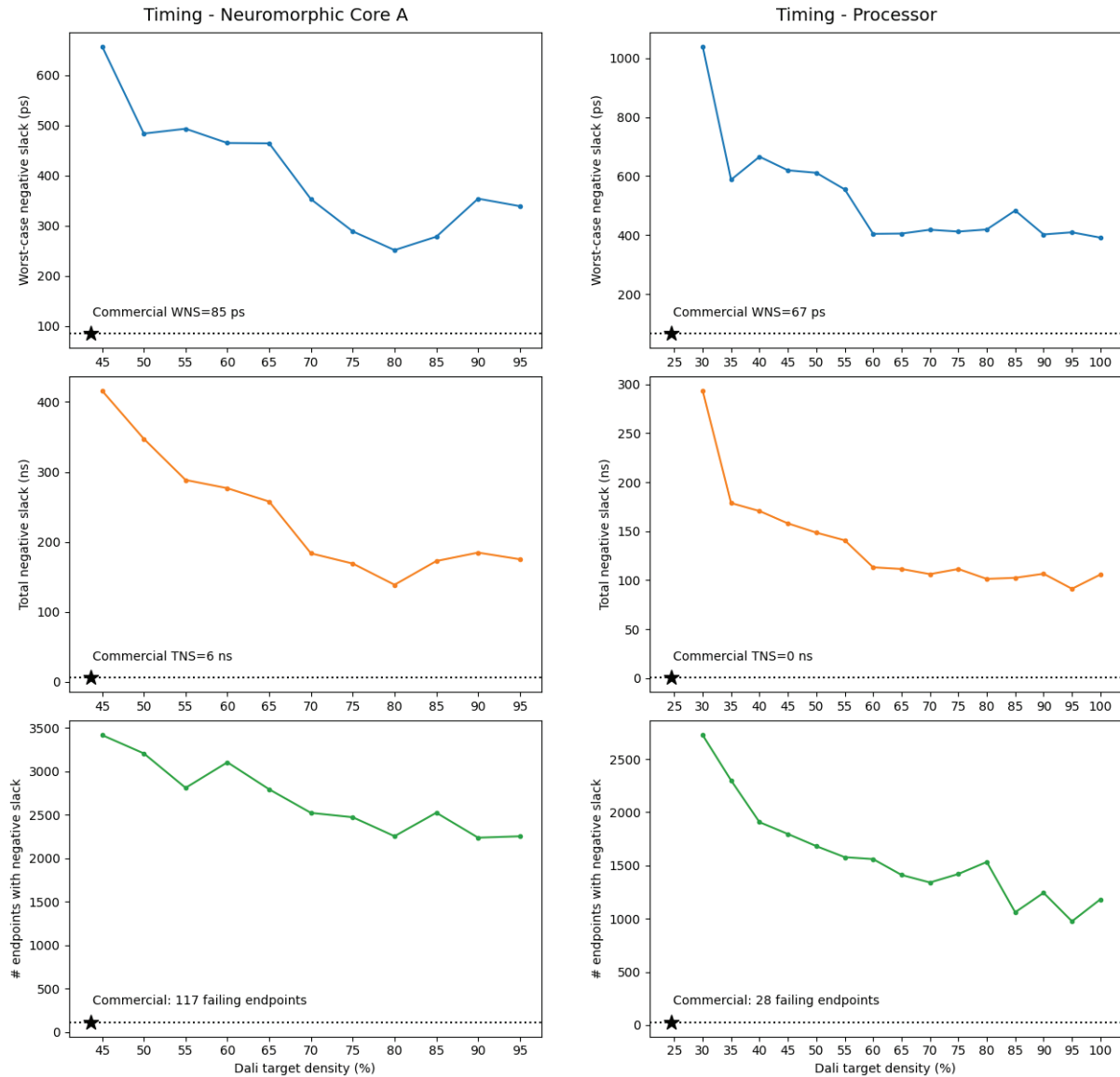


Figure 28: Neuromorphic Core A and Processor - timing Slack

Table 9: Neuromorphic Core A and Processor - timing Slack Summary

	Neuromorphic Core A			Processor		
	Commercial	Dali	ratio	Commercial	Dali	ratio
<i>WNS (ps)</i>	85	251-658	4.3 - 7.7x	67	391-1041	5.8 - 15.5x
<i>TNS (ns)</i>	6	139-416	23 - 69x	0.47	91-293	193 - 623x
<i># failures</i>	117	2237-3417	19 - 29x	28	975-2726	35 - 97x

Figure 28 and Table 9 summarize timing performance for two of the most complex benchmarks: Neuromorphic Core A (left), and Processor (right). Note that for these designs the commercial

flow does not quite meet timing at the placement stage. This would be corrected via timing ECO in the later stages of the flow.

The Processor benchmark timing results are particularly bad for low Dali target density. The same was also true for HPWL (Figure 20). Examining the layout, the Processor floorplan is dominated by fixed hard macros. This means that the available area for cell placement is fragmented. When Dali spreads out the placeable cells to achieve low density throughout the design, the cells spread farther than they would in a floorplan without blockages, and the resulting routes are longer. As Dali begins to compact the cells at higher target densities, this effect is reduced.

The timing performance gaps summarized in Table 9 would require significant timing ECO effort to correct. They indicate that unsurprisingly, simply optimizing wirelength during placement is not adequate to meet timing for complex designs. Timing must be considered directly during the placement task, and this is the subject of the next major chapter of this report.

3.3.2.2 Blockages and Placement Regions

In this section we discuss several advanced placement features required to support our actual neuromorphic designs.

Dali originally supported only rectangular floorplans, but we regularly use designs that include notches to fit with other physical partitions. Figure 29 shows one such example with a non-rectangular floorplan.

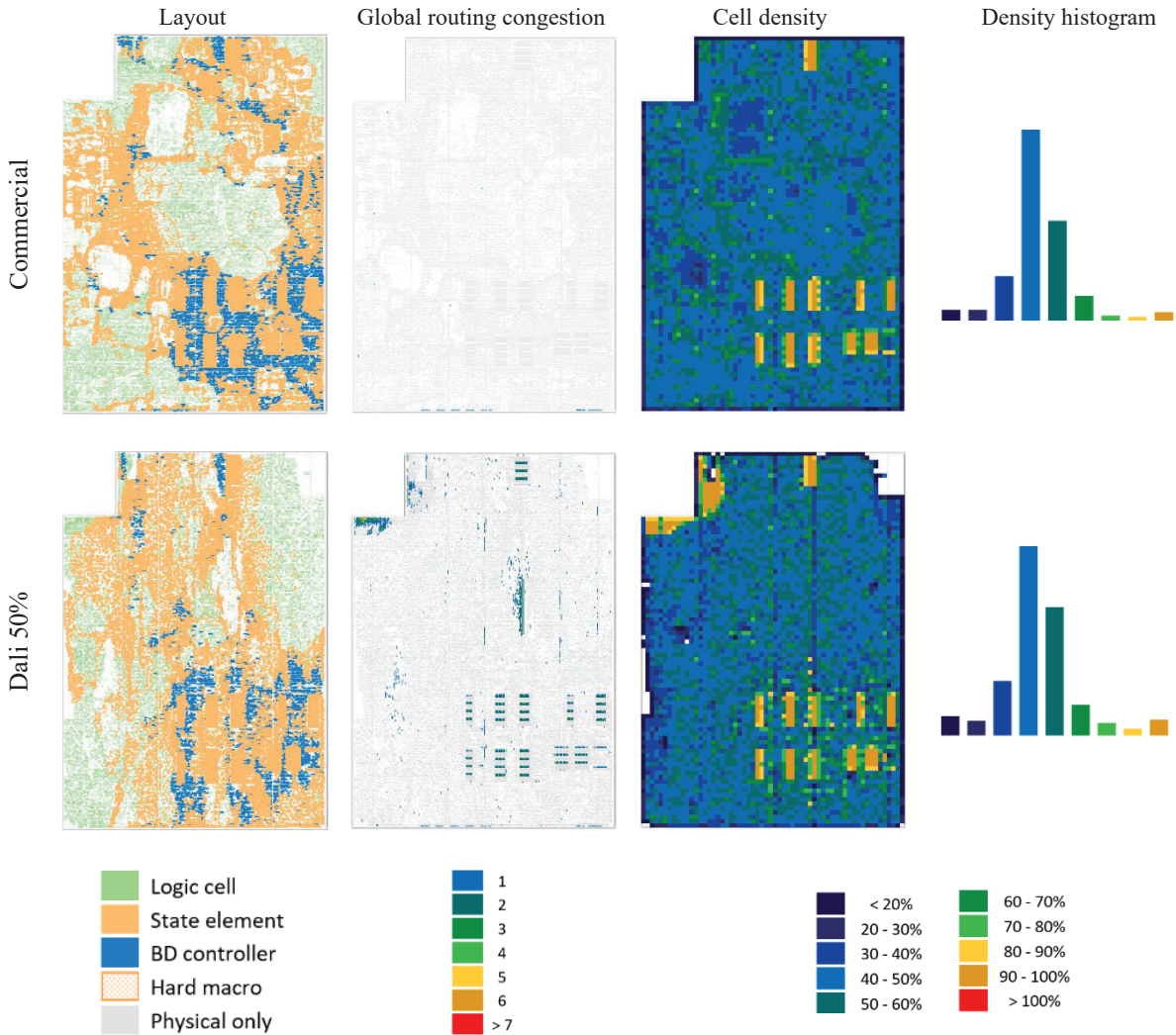


Figure 29: Neuromorphic Core B - layout with Non-rectangular Floorplan

One possible technique for placing designs like this is to temporarily use the circumscribed rectangular floorplan for placement, then legalize the design after the fact by moving any cells placed into the notch. This works passably, but the Dali density and congestion results in Figure 29 show the negative impact of the displaced cells.

Based on our feedback, non-rectangular floorplans are now supported by Dali (implemented as the circumscribed rectangle with a placement blockage covering the notch area). As a result, the placement legalization hack described above is no longer necessary.

One of our most highly structured benchmark designs is the Router. In addition to using large numbers of pre-placed cells, the Router floorplan uses placement blockages over those pre-placed cells to reserve space for routing. It also uses placement regions (Figure 30 (b)) to restrict cells associated with each symmetric half of the Router to the correct side of the floorplan.

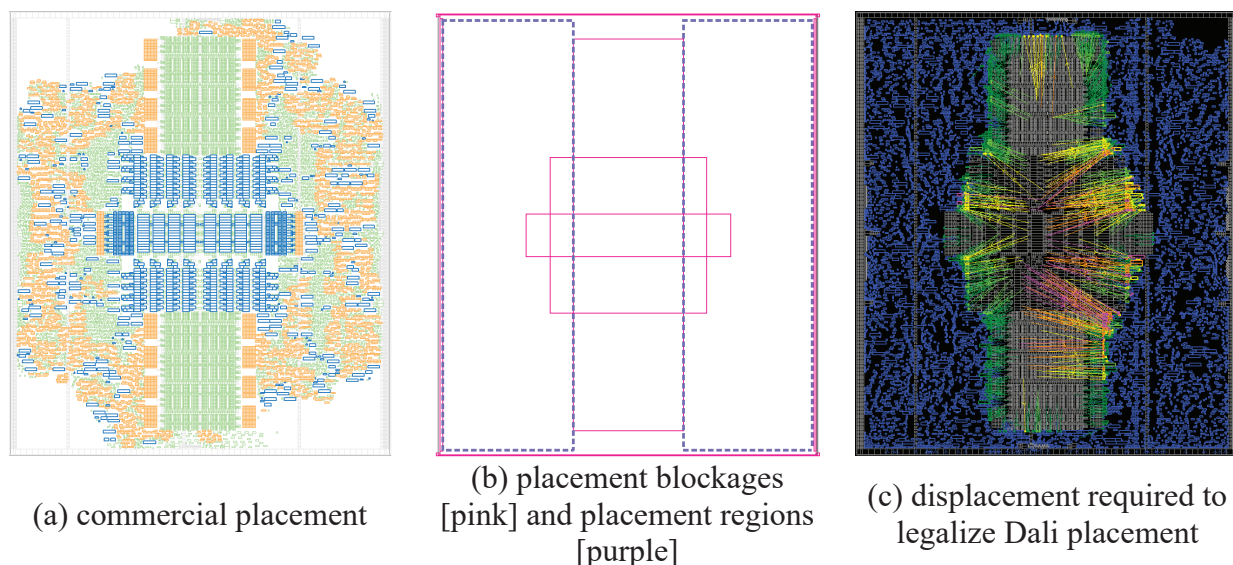


Figure 30: Router Placement with Blockages and Regions

Figure 30 (c) shows the result of first allowing Dali to place the Router on a rectangular footprint with no blockages, and then legalizing the resulting placement to respect the blockage and regions. Blue arrows represent minor displacements, mostly for shifting Dali-placed cells into legal SITES. Green and yellow arrows are medium cell moves, while pink arrows are the largest displacements.

With the addition of Dali support for placement blockages, most of these large legalization displacements are no longer required.

Placement regions are currently not a feature supported by Dali. Studying the legalization map in Figure 30 (c), we can see that almost no displaced cells crossed the design midsection. This means that for this design, Dali largely placed cells on the correct side, even without the benefit of manually defined placement regions. Still, placement regions can be a useful tool for controlling aspects of placement, and would be a welcome enhancement to Dali.

3.3.2.3 Compute Resource Usage

EDA for VLSI layout consumes enormous amounts of compute resources, with run times measured in days or weeks and memory footprints stretching into hundreds of gigabytes. Any significant reduction in resource needs can lead to immediate productivity improvements for design teams by enabling faster turnaround times, more parallel runs, larger physical partition sizes, or all three.

Each data point below represents a single placement trial. For the commercial tool, we consider only time and (maximum) memory consumed by the placement phase.

For Dali, a separate trial is run for each target density, and the minimum and maximum resource usage is reported. Time is the sum of global placement and legalization, although global placement takes so much longer that legalization time is nearly insignificant.

Table 10: Placement Runtime Comparison

Benchmark	Placement runtime (s)			
	Commercial	Dali (min)	Dali (max)	Speedup
Multiply-accumulate [25%]	405	1.4	4.4	91 - 280x
Multiply-accumulate [75%]	286	2.0	2.7	104 - 139x
Floating-point multiply-add [40%]	669	6.6	12.6	53 - 100x
Floating-point multiply-add [60%]	1048	9.3	13.0	80 - 113x
Floating-point multiply-add [70%]	648	10.0	13.2	49 - 64x
Floating-point multiply-add [90%]	1067	7.8	12.7	84 - 136x
Neuromorphic Core A	5551	25.3	38.2	145 - 219x
Processor	2047	14.6	34.1	60 - 140x
Average				78 - 136x
<i>Global placement only (no legalization)</i>				
	Commercial	Dali (min)	Dali (max)	Speedup
Neuromorphic Core B	7385	65.9	86.2	85 - 112x
Router	3512	11.9	29.9	117 - 294x

The largest designs clearly highlight the room for potential improvement: Neuromorphic Core A and B take 1.5 and 2 hours respectively just to complete this stage of placement in the commercial tool, while Dali can be more than 100x faster.

Dali cannot currently be directly compared with the commercial tool based on feature set. It more akin to the inner loop of a larger placement algorithm. These runtime results demonstrate that there is plenty of headroom to re-run Dali iteratively to implement more complex features.

Table 11: Placement Memory Usage Comparison

Benchmark	Placement memory usage (MB)			
	Commercial	Dali (min)	Dali (max)	Reduction
Multiply-accumulate [25%]	6272	33	68	92 - 190x
Multiply-accumulate [75%]	6268	43	53	118 - 145x
Floating-point multiply-add [40%]	6288	82	127	49 - 76x
Floating-point multiply-add [60%]	6298	82	106	59 - 76x
Floating-point multiply-add [70%]	6294	90	101	62 - 69x
Floating-point multiply-add [90%]	6295	97	100	62 - 64x
Neuromorphic Core A	8259	1127	1608	5.1 - 7.3x
Processor	7320	797	1440	5.1 - 9.2x
Average				36 - 51x
<i>Global placement only (no legalization)</i>				
	Commercial	Dali (min)	Dali (max)	Reduction
Neuromorphic Core B	8608	1919	2982	2.9 - 4.5x
Router	7896	1672	4260	1.9 - 4.7x

As with runtime, it is difficult to directly compare Dali memory usage with the commercial flow based on features, since the commercial flow is doing much more. For example, it needs to load libraries for all standard cells for all timing corners, whereas Dali works on only a single corner at a time and loads only the cells used in the design.

Given this fixed overhead borne by the commercial tool, the memory advantage for Dali is much greater for the smallest designs. Still, these results offer some hope of a modest improvement in memory consumption even for the largest designs.

Table 12: Compute Resource Usage Summary Comparison

Benchmark	Basic design statistics				Dali advantage	
	Movable	Fixed	Nets	Utilization	Runtime	Memory
Multiply-accumulate [25%]	881	352	999	21.5%	91 - 280x	92 - 190x
Multiply-accumulate [75%]	869	178	989	72.5%	104 - 139x	118 - 145x
Floating-point multiply-add [40%]	8185	834	8594	37.9%	53 - 100x	49 - 76x
Floating-point multiply-add [60%]	8297	596	8700	57.5%	80 - 113x	59 - 76x
Floating-point multiply-add [70%]	8511	582	8891	71.1%	49 - 64x	62 - 69x
Floating-point multiply-add [90%]	8262	566	8670	89.9%	84 - 136x	62 - 64x
Neuromorphic Core A	68136	7758	74062	43.6%	145 - 219x	5.1 - 7.3x
Processor	29922	27840	37087	24.4%	60 - 140x	5.1 - 9.2x
Average					78 - 136x	36 - 51x

<i>Global placement only (no legalization)</i>						
	Movable	Fixed	Nets	Utilization	Runtime	Memory
Neuromorphic Core B	119200	18372	126723	47.3%	85 - 112x	2.9 - 4.5x
Router	8537	16033	21654	24.1%	117 - 294x	1.9 - 4.7x

3.3.2.4 Suggestions for Improvement

Based on our experience using Dali with a variety of complex bundled-data asynchronous designs, the following set of suggested features would improve its performance and usability.

Timing-driven placement

To achieve feature parity with the commercial tools, Dali needs to be aware of timing constraints in the design. This feature is currently under development and is the focus of the next section of the report. Ideally, the Yale flow would also be able to automatically correct timing violations during the placement process, through e.g. gate sizing, buffer insertion, and delay line insertion. This more ambitious goal builds upon the basic timing-driven placement capability.

Congestion-aware placement

HPWL-based optimization alone leads to the conclusion that denser is better. Unfortunately, denser placements lead to routing congestion later in the flow. Adding (global) routing estimation during placement would allow Dali to avoid causing routing hot spots and lead to better overall QoR.

Placement blockages

Placement blockages are used to prevent cells from being placed where they are not allowed, such as in floorplan corners or along MACRO edges which may violate design rules. Based on our feedback, this capability is now supported by Dali. We use partial placement blockages to limit the percentage of cells placed in a specified area, or to allow only certain cells/sub-designs to be placed in certain areas. This kind of targeted placement blockage can help the tool to have

better routing results, and also reserve space for inserting optimization buffers or ECO cells later to improve the design QoR.

Placement regions

We also use regions to group related cells nearby. For example, grouping sequential cells driven by the same clock source can reduce the number of clock buffers inserted in the clock tree synthesis stage. This is also an opportunity for the Yale flow to go beyond the limited region capabilities typically allowed by commercial tools (i.e. place these cells within a box defined by these coordinates) to support features like relative placement (i.e. keep these cells together, but place them wherever makes sense given other design constraints). This would improve design QoR while reducing the amount of manual floorplan work required of designers.

Non-default rule placement

Non-default rules define special placement rules for library cells based on process technology requirements. Some library cells have certain spacing requirements. For example, 1-fin transistor cells cannot be placed next to non-1-fin cells; always on cells need to have enough space between non-always on cells. These kinds of design rules need to be considered during placement stage.

Easier effort tuning

Dali placement is (for example) 100x faster than the commercial tool, but with 10% worse HPWL. It would be useful to have more user-tunable settings (with sensible defaults) that trade runtime effort for QoR. This would allow for richer comparisons and for the design space to be more thoroughly explored.

3.3.3 Static timing analysis

Static timing analysis is a technique for measuring the performance and timing correctness of a circuit. It requires the design netlist, as well as timing libraries for each cell in the design characterized at a desired operating point (i.e. voltage, temperature, process corner). By “static” we mean that the analysis is carried out without simulation of the circuit in question, so in cases where the timing is data dependent, the results will necessarily be worst case.

3.3.3.1.1 Bundled-data asynchronous design

Bundled-data (BD) is an asynchronous design style where the handshake signals are bundled with the data-path. The handshaking is generally done through separate request and acknowledgement wires as shown in the diagram below.

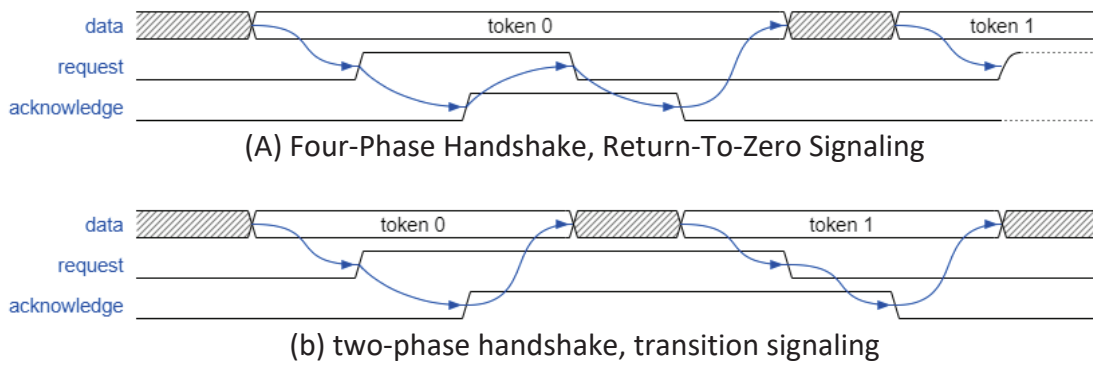


Figure 31: Bundled-data Channel Handshakes

Data must be valid before request is asserted and must remain valid until the acknowledge is received from the process at the other end of the channel. This timing assumption must be guaranteed by the tool flow; bundled-data circuits are not delay-insensitive.

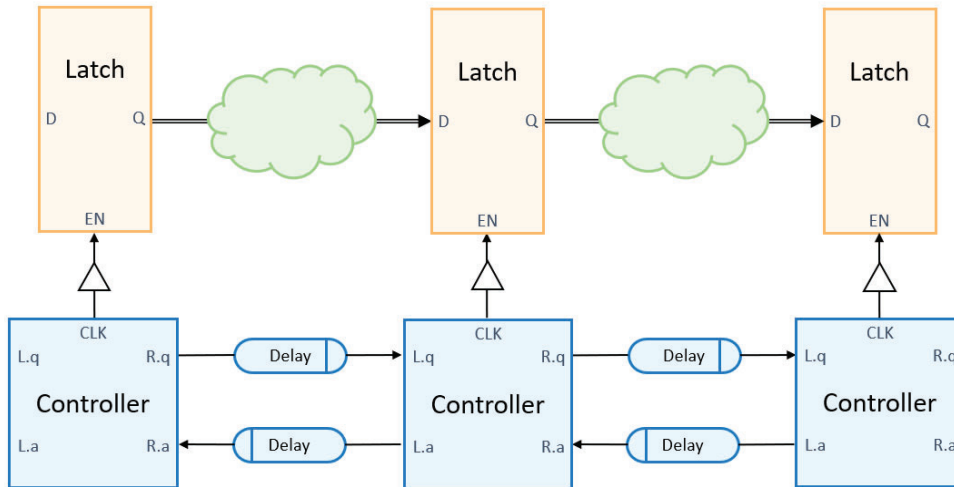


Figure 32: Bundled-data FIFO Pipeline

Each new value on the data path is accompanied by a new request (L.q). At this point, the controller cell latches the data (by pulsing the CLK) and propagates the request forward (through R.q) to the next stages in the pipeline, while also acknowledging the previous stage (through L.a) that it has consumed the data and is ready for a new data value. The controller cannot consume the new data values unless it receives acknowledgment from the next stage in the pipeline (R.a), that it no longer needs its old data.

As seen from the diagram above, a bundled-data design is quite similar to a synchronous design where the global clock is replaced by local clocks that generated by the controller cells. A particular advantage is that combinational logic (shown as green clouds) is exactly the same as in synchronous systems and does not require e.g. delay-insensitive coding. That said, the handshake-logic between the controller cells result in combinational loops, which cause conventional synchronous static timing tools to fail.

3.3.3.1.2 Circuit timing constraints

In circuit design styles that rely on timing assumptions (synchronous or bundled-data asynchronous circuits for example), the constraints that must be satisfied to uphold those timing assumptions derive from the electrical characteristics of the circuits themselves.

For example, Figure 33 depicts the latch, a basic state-holding sequential element used in bundled-data (BD) circuits. Briefly: when EN is high then changes to D are transmitted to Q. When EN is low, Q holds its previous value, regardless of changes to D.

There are three fundamental timing constraints that make this behavior possible:

1. **setup**: D must have a stable value some time before EN transitions high
2. **hold**: D must keep a stable value for some time after EN transitions low
3. **pulse width**: EN must remain high for a minimum amount of time before transitioning low

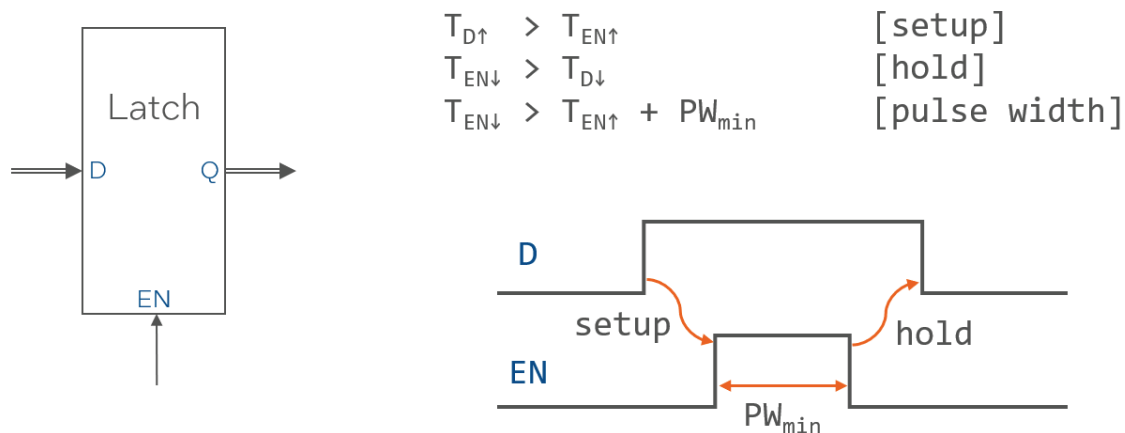


Figure 33: Latch Timing Constraints

If any of these constraints are violated, the value of the latch output Q is not predictable and the circuit fails. Circuits using flip-flops as sequential elements have the same set of timing constraints, plus an additional minimum pulse width constraint for the low phase of the clock signal.

Synchronous timing tools are designed to ensure this set of constraints are met. They are built around the concept of a clock (which would be connected to EN in our latch example), and they track the transition times of other signals in the circuit in relation to transitions of the clock.

3.3.3.1.3 Timing Graph and Ticks

The Yale timer is more generic and does not presume the existence of a governing clock signal. The core of the timer is the timing graph, which is based on the model of (repetitive) event rule systems [27]. The basic components of a timing graph are nodes called events (rising or falling transitions of the output of a gate), and the edges between nodes which represent causal relationships between events and their timing behavior.

Figure 34 shows (a) an example circuit, and (b) the set of events that circuit can generate. These events cannot be ordered in time since $A\downarrow$ appears twice (cannot be both first and last). The solution is to introduce an iteration count as in (c). The timing graph is marked with a **tick** on the edge that crosses an iteration boundary (shown in purple).

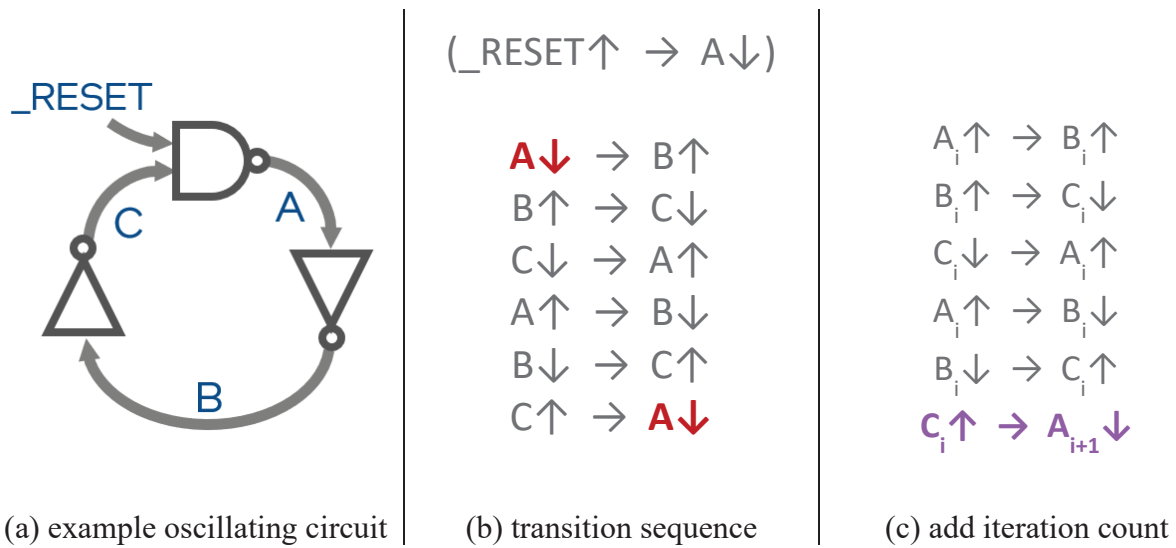


Figure 34: Timing Graph Tick Example

Every cycle in a timing graph must have a tick. A bundled-data handshake is a cycle in the timing graph. By convention, we choose to insert a tick on the acknowledge signal, that is the edge from the successor controller's $L.a\uparrow$ to the predecessor controller's $R.a\uparrow$ (see Figure 31 and Figure 32).

The Yale timer can construct a timing graph from a black box gate-level netlist by assuming that for each gate, every input transition has an edge to every output transition. The timer then uses the Liberty timing libraries to prune the timing graph to include only the arcs explicitly specified in the timing library. For example, an inverter would have an arc: $in\uparrow \rightarrow out\downarrow$, but not the arc: $in\uparrow \rightarrow out\uparrow$.

It is left to the user to add a tick to each cycle in the timing graph in a consistent manner.

3.3.3.1.4 Timing forks

The other core construct of the Yale timer is the timing **fork**. A fork relates the timing of three event (i.e. signal transitions).

root : fast < slow

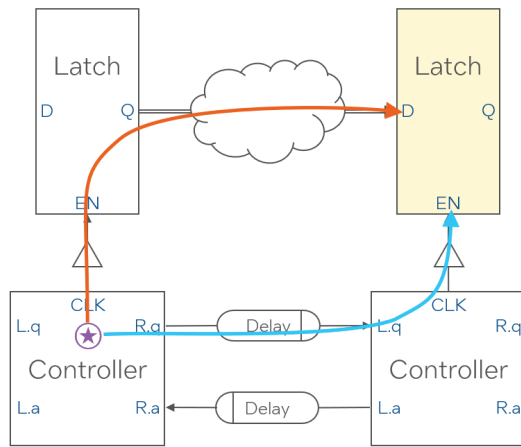
means that after the root signal transition occurs, fast must happen before slow. Timing forks can optionally specify an absolute margin.

Figure 35 depicts how to translate circuit timing constraints for the latch highlighted in yellow into timing forks appropriate for a two-phase bundled-data asynchronous circuit.

For example, the setup timing fork (a) starts with the previous controller. The fast path raises EN on the previous latch which may change D for the latch in yellow. The slow path asserts the request handshake signal, which causes the right controller to raise EN for the yellow latch. The fast path must occur before the slow path, so that the latch circuit setup constraint $T_{D\uparrow} > T_{EN\uparrow}$ is satisfied.

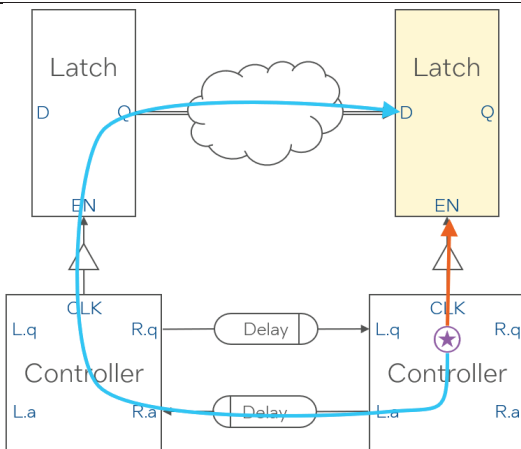
We've added one additional feature to Figure 35: an internal join node for the BD controller used as the root for timing forks and marked with $\star$. This join node represents the union of the controller's L.q and R.a, and signals that it is time to pulse CLK. It is possible to construct a set of timing constraints without the $\star$ node, but all forks would need to be duplicated to refer to both channel inputs. Based on our feedback, the Yale timer was enhanced to support internal nodes so that this optimization is possible.

Note the interaction of the hold fork (b) with the handshake tick on the acknowledge signal. Because we cross the tick when the slow path of the hold fork causes L.a $\uparrow$, that constraint must be tagged with a * to indicate that it crosses a tick boundary. The other hold constraint that causes L.a $\downarrow$ must not have a *.



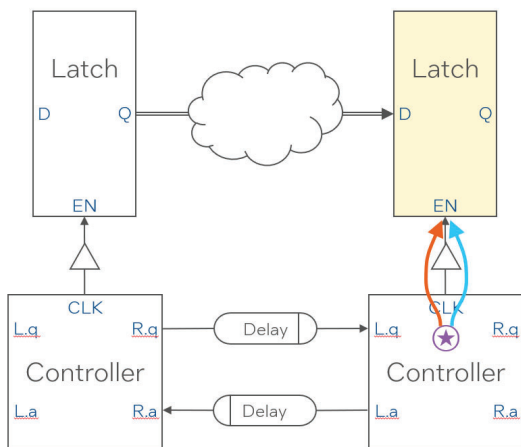
$$\text{prev.} \star : D < EN \uparrow$$

(a) Setup



$$\star : EN \downarrow < (*)D$$

(b) Hold



$$\star : EN \uparrow < EN \downarrow$$

(c) Pulse width

Figure 35: Setup, Hold, and Pulse Width Constraints Expressed as Timing Forks

3.3.3.1.5 False Timing Paths

When applied to our BD designs, the timing forks described in Figure 35 largely operate as designed. Unfortunately, there are a few cases where the timer fails to identify the correct timing path in the circuit. These are shown in Figure 36.

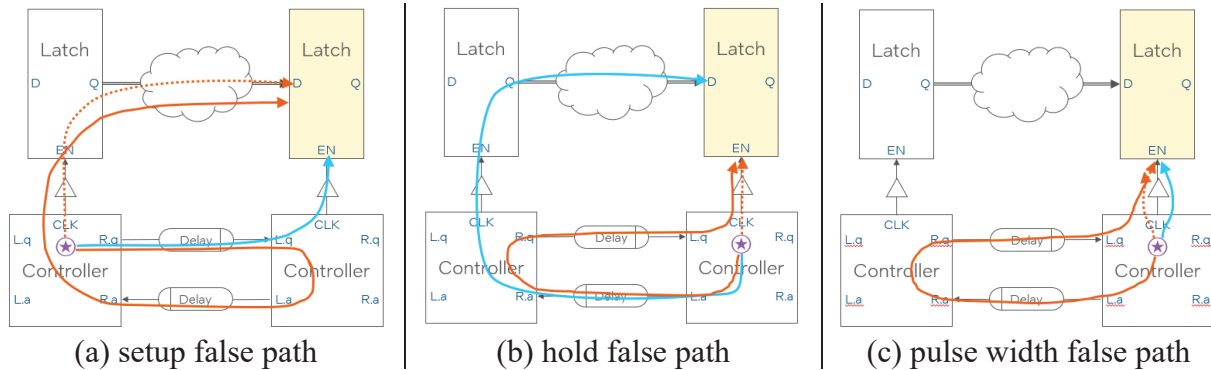


Figure 36: False Timing Fork Paths in FIFO

For each of these false paths, the intended path is shown in dotted orange, while the path the timer chose is in solid orange. Rather than a controller driving CLK directly, the false paths take a round trip across the channel handshake to the neighboring controller first.

The Yale timer looks for the slowest fast path, and the false paths shown do exist in the timing graph. The problem has to do with the tick on the handshake signal $L.a\uparrow$. Because of this, one constraint in each pair will have a false path including $L.a\downarrow$, while the other will be “blocked” by the tick. It is not possible to tick both $L.a\uparrow$ and $L.a\downarrow$, so this appears to be a problem for two-phase BD controllers.

In the Intel Labs flow, a common point $\star$ is defined within the controller cell to start the setup and hold timing races. As such the timing library of the controller represents arcs from these internal nodes to/from the output/input pins of the controller. This is a very common practice used in complex library cells. As the Yale timer didn’t support internal nodes initially, we moved the timing fork root to the controller boundary and created a new timing library. In this new timing library, all timing arcs started and ended on the pins of the controller. With the new arcs in this boundary timing library, we found that the Yale timer saw feed through paths propagating across multiple controllers, which leads to an explosion of paths and is unnecessary for correctness. We tried to use the tick commands to tell the tool that subsequent transitions on the R.q and L.a pins start the next cycle of the timing graph. Unfortunately, our attempts to guide the tool to avoid tracing through multiple controllers did not work. The Yale timer would quit with a failed timing propagation message.

After the Yale timer was upgraded to support internal nodes in the timing library, slow paths in both setup/hold timing check are propagated correctly through the controller’s request/acknowledge pins. However, in certain scenarios, Yale’s timer could not trace the timing graphs completely and would return a “Warning: empty path” message. In setup timing checks,

some flops could not be checked even though other flops driven by same controller's CLK pin are checked correctly. The fast path of setup check related to unchecked flop is reported as an empty path. We confirmed with manual inspection that the fast path exists and has correct timing checks in the commercial tool. For hold timing checks, only the path between first two controller stages is checked correctly. All other hold checks are returned with empty path message for the slow path part as shown blue dotted line in Figure 37. The slow paths of other stages have the same request/acknowledge handshake as the first two stages. Those empty path issues must be fixed to obtain correct timing report.

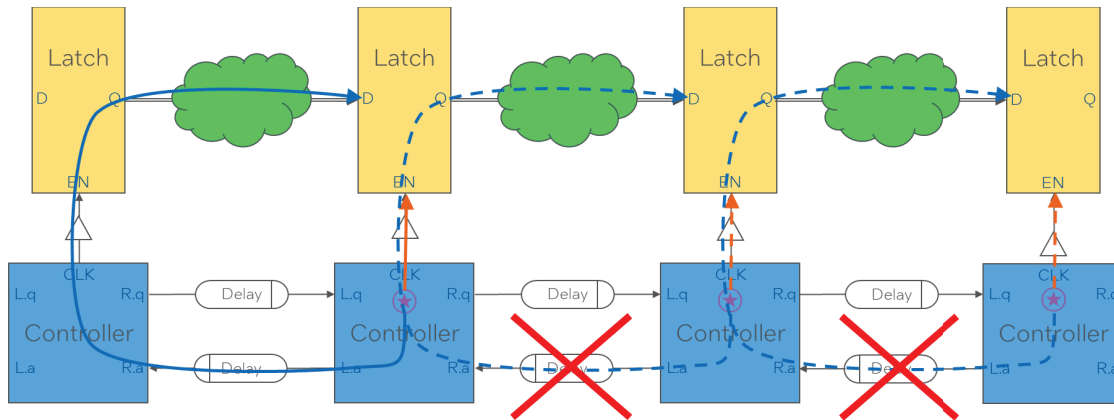


Figure 37: Incorrectly Reported Timing Path

3.3.3.1.6 Future Directions

Timing-driven placement was a work-in-progress at the time of evaluation. Timing-driven placement as implemented by the Yale tools can be thought of as just one iteration of the inner loop of a broader algorithm like those implemented by commercial tools. On top of that foundation, we need to add:

- timing with estimated parasitics/SPEF, to accurately model the true circuit delays
- timing fixes: delay line insertion, gate sizing for delay + slew; clock tree synthesis
- feedback from routing to placement, e.g. congestion based on global placement
- detailed, technology-specific placement rules

3.3.3.1.7 Suggestions for Improvement

Based on our experience using the Yale timer with basic bundled-data asynchronous designs, the following set of suggested features would improve its performance and usability.

Timer API changes

- Single transition tick (add virtual buffer)
- Latency/set_max_delay equivalent
- Utilities for dealing with primary inputs/outputs

- Improved constraint reporting (match with input, warn on duplicate/invalid)
- Automatic tick insertion/suggestion, and/or better examples, especially for complex designs with algorithmic loops. As of the date of this report, this feature has been added to the Yale timer.

Slew rate fixing

Yale tool does not have a convenient way to set slew rate limit for whole design and for a certain part of design. Normally, clock path has smaller slew rate limit. In our BD design, clock trees from controller's CLK pin and request/acknowledge paths have smaller skew rate requirement. In commercial tools, since we define clocks, we can easily set slew rate limit for those clocks, the commercial tool will apply slew limit and propagate to entire clock and request/ack paths. In Yale's tool, the slew limit needs to be defined for each net.

Moreover, Yale tool only has commands to size cells and insert buffers manually. It does not automatically size up cells or insert buffers to fix any skew violation.

Clock tree synthesis

In our BD design, we define local clocks at controllers' CLK pin which connect to sequential cells. We do clock tree synthesis in commercial tool to balance clock trees for sequential cells for better timing result. Yale's tool does not have clock concept and it does not have a command to create a balanced tree from a certain node.

Delay insertion

The Yale timer only reports timing slack violations; it does not suggest where to insert delay to resolve them. This is intended as future work (see the << operator in the timing language body), but currently must be scripted manually.

Library cell selection

We have many different library cells. Their logic function may be the same but can be used for different purposes. For example, regular buffers' function is equivalent to clock tree buffers. However, we want to use clock tree buffers for clock path only. Some library cells may have more leakage but better timing (low threshold voltage cells). When we consider timing quality and power consumption, we limit the percentage of low threshold cells to balance the power consumption and timing quality trade-off.

Advanced timing libraries

Yale's timer currently only supports NLDL libraries, not more recent timing models (e.g. CCS, LVF). These advanced models are better able to represent the statistical variation present in advanced process technologies.

3.3.4 Summary

The Intel study provided valuable feedback in terms of what features should be prioritized for future development. Most of the issues that arose were not surprising; for example, the placement engine we developed uses both local and global density constraints, which results in the "spread out" layout under low densities. While congestion information is computed by our global router and written to the physical database, this information is currently not used by the placement engine. Finally, there were many issues with using our static timing analysis engine because the Intel internal flow uses commercial timers; hence, the timing graph used by Intel is acyclic. This means that the timing graph does not directly correspond to the asynchronous circuit, and hence the core cyclic timing analysis and timing propagation from cycles could not be used. We made a number of changes to our timer to support the types of graphs being generated by Intel, but more work remains.

3.4 Transitions

Based on questions / requests for clarification that we have received, the ACT tools are currently being used by over a two dozen academic groups and at least ten companies, including three major semiconductor corporations. One former Ph.D. student supported partially by this project has launched a startup company to commercialize their thesis research on asynchronous design, while a second student is currently exploring startup formation.

The tools are currently in use by two Department of Energy national labs (Lawrence Livermore and Brookhaven). We are currently working collaboratively with Lawrence Livermore National Labs on a joint project, and the funded project is supporting further development of the ACT tools and creating libraries for multiple asynchronous circuit families.

The National Science Foundation funded an efabless.com ChipIgnite tape-out for the open-source Skywater 130nm process as part of the 2022 Summer School on Asynchronous Design. Selected designs completed by participants will be submitted for fabrication in the Spring of 2024.

We presented a tutorial at the Telluride Neuromorphic Cognition Engineering workshop in 2023 with the goal of making the ACT tools accessible to the neuromorphic computing research community. We also presented tutorials at two mainstream circuit venues: the 2023 IEEE International Symposium on Asynchronous Circuits and Systems, and at the 2023 IEEE European Solid-State Circuits Conference.

The ACT tools are also an integral part of projects in the DARPA FENCE program as well as the DARPA QUICC program. Our work on QUICC will use the ACT tools for a mixed-signal chip that contains both custom analog components and digital components specified and implemented in the ACT framework.

A ~\$4M project to develop an implantable, low power asynchronous vector processor has been rated above the National Institutes of Health (NIH) payline. This effort is centered on using the ACT tool flow for chip development and should start in April 2024. This will also fund further development of front-end ACT design tools for logic synthesis and automated pipelining.

Finally, Yale is part of the NORDTECH Microelectronics Commons Hub, and we are currently developing a project proposal on chiplets for artificial intelligence hardware that would not be possible without the ACT EDA tools.

3.5 Summary of Open-Source Software Tools Developed

The software tools developed as part of this project have been released via an open-source license and are available on the public Github site at the following location:

<https://github.com/asynclsi/>

An integrated repository that builds and installs all the key software tools is

<https://github.com/asynclsi/actflow>

The following table summarizes major tools developed and released over the course of this project.

EDA Tool	Description
prsim	Async simulation + hazard detection
lvp	Layout verification v/s ACT
prs2net	Hierarchical SPICE netlist from ACT description
prs2cells	Netlist to cell mapping
aflat	Flatten design for simulation
act2v	Export ACT to Verilog netlist
ext2sp	Convert magic extract file to a hierarchical SPICE file
prs2sim	Convert ACT to sim format (for open source IRSIM simulator)
v2act	Import Verilog netlist into ACT; sync to async import

tlint	Signal integrity validation
AMC	Asynchronous memory compiler
dali	Gridded cell placer
PWRoute	Power detailed router
SPRoute	Parallel global router
BiPart	Parallel hypergraph partitioner
cyclone	Parallel asynchronous static timing and power analysis
act2lef	Cell mapper + generator
xcell	Cell characterizer
interact	Integrated tool flow for the implementation flow

4 CONCLUSIONS

We developed the first open-source EDA flow for asynchronous logic. The logic flow was generic, in that it was designed to support any asynchronous logic family. The flow was successfully used to fabricate several different chips. In addition, the flow was evaluated by Intel and compared to state-of-the-art commercial EDA tools that have numerous advanced features that are not found in any open-source EDA tools. Several opportunities for improvement / feature enhancements were identified, and the implementation of some of the features is currently underway.

Educational materials and documentation were developed for the tool flow, and we currently have many external users including major semiconductor companies. We ran the first ever summer school on asynchronous design with the support of the steering committee of the IEEE Symposium on Asynchronous Circuits and Systems. Additional funding from the National Science Foundation (NSF) was provided to support this effort, and we had over 300 participants from around the world including academia, industry, and government.

5 REFERENCES

- [1] A. J. Martin, “The limitations to delay-insensitivity in asynchronous circuits,” in *Proceedings of the sixth MIT conference on Advanced research in VLSI*, in AUSCRYPT ’90. Cambridge, MA, USA: MIT Press, 1990, pp. 263–278. [Online]. Available: <http://dl.acm.org/citation.cfm?id=101415.101434>
- [2] R. Manohar and A. J. Martin, “Quasi-Delay-Insensitive Circuits are Turing-Complete.” Department of Computer Science, California Institute of Technology, Pasadena, CA, 1995.
- [3] R. Manohar and Y. Moses, “Analyzing Isochronic Forks with Potential Causality,” in *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, 2015, pp. 69–76.
- [4] I. Sutherland and S. Fairbanks, “GasP: A minimal FIFO control,” in *Asynchronous Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, 2001, pp. 46–53.
- [5] 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*, 2001, pp. 9–17.
- [6] C. A. R. Hoare, “Communicating Sequential Processes,” *Commun ACM*, vol. 21, no. 8, pp. 666–677, 1978.
- [7] R. Manohar, “ACT hardware description language documentation.” 2018.
- [8] R. Manohar and Y. Moses, “Timed signaling processes,” in *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2023, pp. 10–19.
- [9] K. Pingali *et al.*, “The tao of parallelism in algorithms,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 12–25.
- [10] R. Manohar, “xcell: a cell library characterizer for combinational and state-holding gates,” in *Workshop on Open-Source EDA Technology*, 2022.
- [11] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, “Multilevel hypergraph partitioning: Application in VLSI domain,” in *Proceedings of the 34th annual Design Automation Conference*, 1997, pp. 526–529.
- [12] 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*, 2021, pp. 161–174.
- [13] R. Karmazin, C. T. O. Otero, and R. Manohar, “cellTK: Automated Layout for Asynchronous Circuits with Nonstandard Cells,” in *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, 2013, pp. 58–66.
- [14] Y. Yang, J. He, and R. Manohar, “Dali: A gridded cell placement flow,” in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.
- [15] 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)*, 2020.

- [16] M. Pan and C. Chu, “FastRoute: A step to integrate global routing into placement,” in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 464–471.
- [17] 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)*, 2019, pp. 1–8.
- [18] W. Hua, Y.-S. Lu, K. Pingali, and R. Manohar, “Cyclone: a static timing and power engine for asynchronous circuits,” in *IEEE International Symposium on Asynchronous Circuits and Systems ({ASYNC})*, 2020, pp. 11–19.
- [19] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
- [20] S. M. Burns and A. J. Martin, *Performance Analysis and Optimization of Asynchronous Circuits*. 1990.
- [21] S. M. Burns, H. Hulgaard, T. Amon, and G. Borriello, “An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems,” *IEEE TRANSACTIONS ON COMPUTERS*, vol. 44, no. 11, pp. 1306–1317, 1995.
- [22] W. Hua and R. Manohar, “Exact timing analysis for asynchronous systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, no. 1, pp. 203–216, 2018.
- [23] R. Manohar, “Exact timing analysis for asynchronous circuits with multiple periods,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 3134–3138, 2019.
- [24] K. S. Stevens, R. Ginosar, and S. Rotem, “Relative timing,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, no. 1, pp. 129–140, 2003.
- [25] S. Ataei and R. Manohar, “AMC: An asynchronous memory compiler,” in *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2019.
- [26] J. He *et al.*, “interact: An Interactive Design Environment for Asynchronous Logic,” in *Workshop on Open-Source EDA Technology, International Conference on Computer-Aided Design (ICCAD)*, 2021.
- [27] S. M. Burns, “Performance Analysis and Optimization of Asynchronous Circuits,” pp. 1–214, Feb. 1991.

6 LIST OF ACRONYMS

Acronym	Expanded Form
ACT	Asynchronous circuit toolkit, and also asynchronous CAD tools
AMC	Asynchronous memory compiler
ASIC	Application-specific integrated circuit
BD	Bundled data
CAD	Computer-aided design
CHP	Communicating hardware processes
DEF	Design Exchange Format
EDA	Electronic design automation
GDSII	Graphic Design System II
LEF	Layout Exchange Format
NORDTECH	Northeast Regional Defense Technology Hub
NSF	National Science Foundation
IDEA	Intelligent design of electronic assets
IDEAL	Intelligent design environment for asynchronous logic
PRS	Production rules or Production rule set
SoC	System-on-a-chip
SRAM	Static random access memory