Stixu: An Open-Source Idea for Planning Integrated Circuit Layouts Efficiently

1st Nicholas Overacker
Info. & Communication Engineering
Kitami Institute of Technology
Kitami, Japan
m3235380013@std.kitami-it.ac.jp

2nd James Stine
Electrical & Computer Engineering
Oklahoma State University
Stillwater, OK
james.stine@okstate.edu

3rd Michal Ptaszynski
Info. & Communication Engineering
Kitami Institute of Technology
Kitami, Japan
ptaszynski@ieee.org

4th Shingo Yoshizawa
Info. & Communication Engineering
Kitami Institute of Technology
Kitami, Japan
yosizawa@mail.kitami-it.ac.jp

5th Miho Kobayashi
Independent UI Designer
Nagano, Japan
kobayashi@ringodesigners.com

***Abstract*—Stick diagrams are a traditional tool used to quickly design the topology of an integrated circuit on paper before beginning the time-consuming CAD layout process. Stixu is a browser-based, open-source EDA tool for stick diagrams. It provides built-in output analysis capability and offers both VLSI learners and professionals with an open-source means to quickly and easily validate CMOS topologies.**

I. Copyright Notice

II. Introduction

Stick diagrams have been used as an abstraction of integrated circuits since at least the 1970s [1] [2, pp. 2-3]. Although several computer programs have been developed [2, p. 3] [3] and data interchange standards proposed [4] for digital stick diagrams, at the time of submission the authors have not found any publicly available software dedicated to the validation of stick diagrams.

In this paper, we present Stixu, an open-source, mobile-compatible tool designed to make the process of drawing and validating stick diagrams easier and more accessible [1]. Stixu can be used to quickly validate a proposed layout within minutes from any PC, phone, or tablet before implementing it in physical layout software such as *Magic*.

This tool can be run from the official website or locally from the source code on any computer without modification. In addition to its potential to save time for professionals and academics who need to draw, validate, and share stick diagrams, it can also play a vital role for educators and students in a classroom environment as well as for professional VLSI engineers using open-source Electronic Design Automation (EDA) tools.

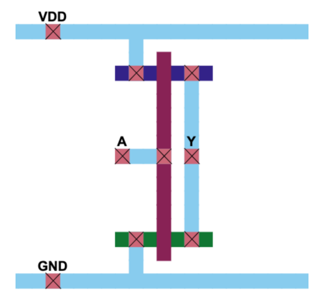

Fig. 1. The stick diagram for a CMOS inverter. The input contact A is connected to a polysilicon ‘stick’ that forms the gate of a PMOS transistor on top and an NMOS transistor on the bottom. The drains of both transistors are connected to a stick that contains the output contact, Y. (Drawn in Stixu.)

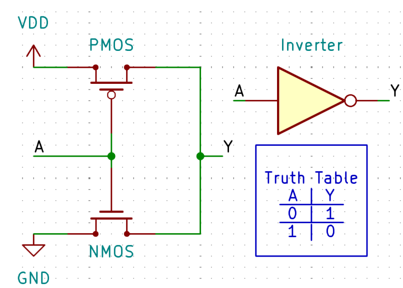

Fig. 2. CMOS and logic circuits for Figure 1. (Drawn in KiCad.)

III. Background: Regarding Stick Diagrams

Stick diagrams are topological representations of digital integrated circuits [5, pp. 28-29]. Because an integrated

[1]Available at https://github.com/NickOveracker/StickDiagrammer and https://stixu.io

circuit is comprised of many layers that are electronically connected in some way, the stick diagram enables an essential methodology for visualizing the layers for the layout engineer to integrate them into an EDA tool. Diffusion, polysilicon, and metal layers are represented as intersecting zero-width lines (or 'sticks'). Layers are typically differentiated by assigning distinct colors to each layer, although they have also historically been implicitly differentiated in monochromatic diagrams by labeling transistors as either P- or N-type [3, p. 290]. It is important to emphasize that stick diagrams are 'planning' tools and do not indicate what the layout looks like. They only provide valuable guidance for the layout engineer in implementing the design within the EDA tool.

Line intersections in the same layer are shorted together, and intersections of different layers are open unless a via is drawn at the point of intersection (typically represented by a dot or cross). Open intersections of the polysilicon layer with p-diffusion or n-diffusion layers create PMOS or NMOS transistors, respectively. P and N diffusion layers cannot intersect. The stick representation of a simple Complementary Metal Oxide Semiconductor (CMOS) inverter [5, p. 28] is illustrated in Figure 1.

Stick diagrams are analyzed as idealized circuits with no complicating factors such as resistance, capacitance, and inductance. In principle, any valid stick diagram can be redrawn as a CMOS circuit schematic, as illustrated in Figure 1. These circuits can then be validated using traditional methods such as SPICE or Hardware Descriptive Language (HDL) simulation [5, pp. 40-45, 53].

Because the process of manually translating hand-drawn stick diagrams to CMOS schematics for analysis can be tedious and error-prone for non-trivial diagrams, a method to directly evaluate their behavior is desirable. Several "stick compiler" and "symbolic layout" systems developed in the decades following the introduction of the stick diagram are described in the literature, including but not limited to STICKS [3], MGX [6], and PSI [7].

These systems however are concerned primarily with the transformation of stick diagrams to physical layouts rather than being the analysis of stick diagrams themselves. Furthermore, they seem to be either unavailable or inaccessible to the general public in any form today. Stixu was developed to fill this gap.

IV. Implementation

The Stixu stick diagramming tool is implemented as an entirely client-side web browser application. In this section, we will briefly discuss the major components of the software implementation, and we will pay particular attention to the circuit model and evaluation algorithm.

Users can draw stick diagrams in Stixu by selecting layers from the color palette, clicking (or touching) a starting point on the on-screen canvas, and dragging to draw horizontal or vertical lines. The overall interface and mode of interaction

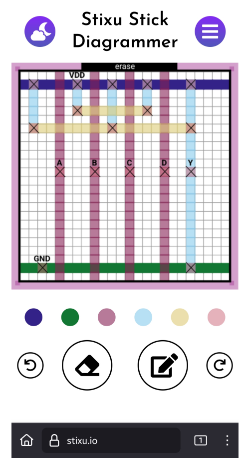

Fig. 3. A four-input NAND gate implemented in Stixu's mobile interface. (Rendered in Firefox for Android.)

was largely inspired by the free and open-source VLSI layout program Magic, and is intended to be a familiar experience for anyone who has used computer drawing software such as MS Paint.

To accommodate classroom use and other scenarios where a PC may not be available, implementing a convenient and feature-rich mobile interface (Figure 3) is a top priority. All of Stixu's documented features are available in the mobile interface, and it is a guiding principle of the project that all future desktop features should also be available on mobile.

The PC interface (Figure 4) offers convenient keyboard shortcuts, but there are no PC-only features except for undocumented key-bindings whose graphical interface have not yet been implemented. In fact, the mobile interface can be rendered on PC and vice versa simply by adjusting the aspect ratio between portrait and landscape orientations.

The default color scheme, based on Paul Tol's 'muted' palette, is chosen to be inclusive for users with color-vision deficiency (CVD) [8, p. 16] as well as familiar to those who use the VLSI layout program Magic. Non-diffusion layers are rendered with transparency by default in order to distinguish between intersections and tangents, but layer transparency can be turned off to provide greater contrast, as seen in the diagram in Figure 1.

Users generate a truth table for their stick diagram by clicking or tapping the 'Evaluate' button, which can be found in the mobile interface by scrolling down and is in view by default on PC. This can be used immediately validate the correctness of a design, as shown in Figure 4.

Possible output values currently include 0 (logic 0), 1 (logic 1), Z (high impedance), X (short), L (unstable logic 0), H (unstable logic 1), and U (unknown). H, L, and U occur in circuits whose input-to-output paths include floating gates or memory elements. Clicking or tapping a particular cell in the output column of the table will highlight all nets that are connected to the output for a given input vector,

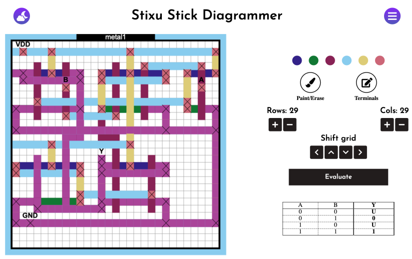

Fig. 4. A NAND D flip flop with the output path for input vector 1 (i.e., [A,B] = [0,1]) highlighted. (Rendered in Firefox for MacOS.)

as shown in Figure 4. This feature can help users improve designs by showing how incorrect values are produced.

The netlist is implemented as a list of sets containing mutually-reachable cells. The netlist generated at the start of output analysis basically in accordance with Algorithm 1. Nets are built up starting from each input and output terminal (VDD, GND, all independent inputs, and all outputs) and expand recursively to cover all connected cells.

Where transistors are encountered, new nets are created for each of the terminals other than the one that was already encountered, and the net generation process begins anew for each of those terminals. At the end of this process, identical nets are merged. Islands that cannot be reached from any external input or output terminal are not processed.

During the netlist generation stage, external input and output terminals (including VDD and GND) and all transistor elements are modeled as vertices in a hypergraph. Vertices on the same net are connected by a single hyperedge per net, with additional hidden vertices representing logical 0 and 1. Each input vector is evaluated by connecting VDD, GND, and all inputs to the global logic vertices. Transistor gate vertices are tested for paths to these vertices. Active transistors connect their source and drain vertices with new hyperedges, repeating the loop until no more updates occur. The output is tentatively assigned a value based on paths to the global logic vertices: 1 (path to global logic 1), 0 (path to global logic 0), Z (no path), or X (both paths). If the result is X, it is returned as the final output for the current input vector.

If further evaluation is needed, transistors with floating gates are assigned all possible test signals as shown in as shown in Algorithm 2. This is done in order to determine whether indeterminate paths can result in a short circuit or a valid output. If any path is found from global logic 1 to global logic 0, the output is set to X to indicate a fault. The final output is returned after all iterations have completed or a fault is found.

All updates to the Stixu source code are validated against a 58-case test bench that not only tests common circuits such as D flip flops, 5-stage inverters, and AND gates, but also several different invalid circuits with different failure modes (such as circuits with two independent inputs on the same net) as well as basic user interface functions such as changing the color scheme. The expected outputs were originally determined by visual inspection, but for future test bench cases these will be partly produced by HDL simulation of Verilog code generated by Stixu.

For each tested diagram, all rotations and reflections of the same diagram are also tested to ensure the the absence of asymmetry in the output analysis algorithm. New tests are continuously added to explore edge cases.

V. Results

Early versions of the evaluation algorithm used deep and branching recursion with standard graphs featuring two-node edges. Implementing this correctly proved to be very diffi-

Algorithm 1 The netlist generation algorithm for Stixu.

for terminals t in $\{VDD, GND, Inputs, Outputs\}$ **do**
  $t.net$.EMPTY()
  ADD_TO_NET($t.net, t.cell, contact$)
end for
MERGE_IDENTICAL_NETS()
for all transistors t **do**
  for all $term$ in $\{t.source, t.drain, t.gate\}$ **do**
    $term.net \leftarrow term.net$ || NEW Net($term.cell$)
    ADD_TO_NET($term.net, term.cell, term.layer$)
  end for
end for
function ADD_TO_NET(net,$cell$,$layer$)
  if net.INCLUDES($cell$,$layer$) **then**
    return
  end if
  if $cell$ is an uninitialized transistor **then**
    $t \leftarrow$ NEW Transistor($cell$)
    $transistors$.INSERT(t)
    if $layer$ is a diffusion layer **then**
      return
    end if
  end if
  net.INSERT($cell, layer$)
  if $cell.active\_layers$.INCLUDES($contact$) **then**
    for all $loop\_layer$ in $cell.active\_layers$ **do**
      ADD_TO_NET($net, cell, loop\_layer$)
    end for
  end if
  if $layer = contact$ **then**
    return
  else
    for all $loop\_cell$ in adjacent cells **do**
      if $layer$ is set at $loop\_cell$'s coordinates **then**
        ADD_TO_NET($net, loop\_cell, layer$)
      end if
    end for
  end if
end function

Algorithm 2 The ouput algorithm for Stixu.

```
function COMPUTE(input_vals[ ], output_vtx)
    RELOAD_HYPERGRAPH()
    ADD_EDGE(vdd_vtx, logic_1_vtx)
    ADD_EDGE(gnd_vtx, logic_0_vtx)
    for i in COUNT(input vertices AS input_vtx[ ]) do
        if input_vals[i] = 1 then
            ADD_EDGE(input_vtx[i], logic_1_vtx)
        else ADD_EDGE(input_vtx[i], logic_0_vtx)
        end if
    end for
    loop_again ← TRUE
    while loop_again do
        loop_again ← FALSE
        for all transistors t do
            if t.gate.IS_ACTIVE() then
                ADD_EDGE(t.source, t.drain)
                loop_again ← TRUE
            end if
        end for
    end while
    output ← Z
    if output_vtx.HAS_PATH_TO(logic_1_vtx) then
        output ← 1
    end if
    if output_vtx.HAS_PATH_TO(logic_0_vtx) then
        if output = 1 then output ← X
        else output ← 0
        end if
    end if
    for i ← [0..2^{COUNT(floating_trans[ ])} − 1] do
        for j ← [0..COUNT(floating_trans[ ])−1 do
            if BITWISE_AND(i, 2^j) ≠ 0 then
                // Only join if still floating.
                SOFT_ADD_EDGE(source[j], drain[j])
            end if
        end for
        if output.HAS_PATH_TO(logic_1_vtx) then
            if output.HAS_PATH_TO(logic_0_vtx) then
                output ← X
            else if output = Z then output ← H
            else if output = 0 then output ← X
            else if ouput = L then output ← U
            end if
        else if output.HAS_PATH_TO(logic_0_vtx) then
            if output = Z then output ← L
            else if output = 1 then output ← X
            else if ouput = H then output ← U
            end if
        end if
        REVERT_HYPERGRAPH_CHANGES()
    end for
    return output
end function
```

cult. Deeply-rooted bugs were discovered that manifested as asymmetric results when diagram mirroring and rotations were added to the test bench. This was found to be due to each component only ‘knowing’ which components were immediately adjacent to it rather than which components are electrically connected to it. The algorithm was rewritten with a much simpler and non-recursive hypergraph-based evaluation function to correct this.

In its current state, Stixu successfully analyzes all tested complex topologies, including deeply nested outputs such as multi-stage inverters, as well as recursive feedback loops such as SR latches and D flip flops (see Figure 4). The individual test cases may all be seen and inspected in the Stixu repository [9] or the live test bench page [10].

VI. Next Steps

Planned additions to Stixu include hierarchical designs with saved diagrams, a user-facing implementation of the experimental Verilog generation function, and a system for creating and evaluating assignments in educational settings.

Acknowledgments

The corresponding author wishes to extend gratitude to numerous mentors, including Sam Ball, Wira Mulia, Drs. Carl Latino, Blayne Mayfield, Josiah Meints, John O'Hara, David Lampert, Vignesh Rajamani, Mark Rockley, Peter Shull, and Keith Teague of Oklahoma State University, Dr. Jingtong Hu of the University of Pittsburgh, Dr. Damon Chandler of Ritsumeikan University, Dr. Tomonori Sato of Shinshu University, Jim Wirt and Aaron Bean of Beardon Services, and Hideki Denda of Shinko Electric Industries Co., Ltd., among many others. Special recognition also goes to the contributions of Matt Venn and Uri Shaked to the democratization of IC fabrication and VLSI education, including their work with *Tiny Tapeout* [11] and the educational browser-based layout SPICE simulator *SiliWiz* [12]. Finally, thanks to Sam the Designer for Stixu's future official logo, and to the YouTube education channel *Vital Sine* [13] for inspiring the development of Stixu's current algorithms.

References

[1] J. D. Williams, "Sticks–a new approach to lsi design," Master's thesis, Massachusetts Institute of Technology, June 1977.

[2] H. Watanabe, *IC Layout Generation and Compaction Using Mathematical Optimization*. PhD thesis, University of Rochester, 1984.

[3] J. D. Williams, "Sticks - a graphical compiler for high level lsl design," in *Managing Requirements Knowledge, International Workshop on*, (Los Alamitos, CA, USA), p. 289, IEEE Computer Society, jun 1978.

[4] S. Trimberger, "The proposed sticks standard," Tech. Rep. Technical Report 3880, California Institute of Technology, October 1980.

[5] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. USA: Addison-Wesley Publishing Company, 4th ed., 2010.

[6] M. Terai, Y. Ajioka, T. Noda, M. Ozaki, T. Umeki, and K. Sato, "Symbolic layout system: Application results and functional improvements," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 3, pp. 346–354, 1987.

[7] R.-D. Fiebrich, Y.-Z. Liao, G. M. Koppelman, and E. N. Adams, "Psi: A symbolic layout system," *IBM J. Res. Dev.*, vol. 28, pp. 572–580, 1984.

[8] P. Tol, "Colour schemes." Available at https://personal.sron.nl/~pault/data/colourschemes.pdf, 2021. Accessed: 2024/5/23.
[9] N. Overacker and M. Kobayashi, "Stickdiagrammer." Available at https://github.com/NickOveracker/StickDiagrammer. Accessed: 2024/5/25.
[10] N. Overacker and M. Kobayashi, "Stixu testbench." Available at https://stixu.io/test/. Accessed: 2024/5/25.
[11] M. Venn and U. Shaked, "Tiny tapeout." Available at https://tinytapeout.com/. Accessed: 2024/5/24.
[12] U. Shaked, "Siliwiz." Available at https://app.siliwiz.com. Accessed: 2024/5/24.
[13] "Vital sine." Available at https://www.youtube.com/@VitalSine. Accessed: 2024/5/24.