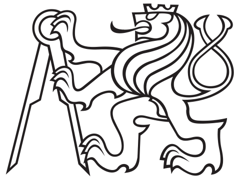
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Radio Engineering

Analysis of Digital and Embedded Engineering Methods for System on Chip Design

Bc. Emil Jiří Tywoniak

Supervisor: Associate Prof. Ing. Stanislav Vítek, Ph.D.
Field of study: Open Electronic Systems
Subfield: RF and DSP Engineering
January 2023

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Tywoniak Emil Ji í** Personal ID number: **474266**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Radioelectronics**

Study program: **Open Electronic Systems**

Branch of study: **RF and DSP Engineering**

II. Master's thesis details

Master's thesis title in English:

Analysis of Digital and Embedded Engineering Methods for System on Chip Design

Master's thesis title in Czech:

Analýza metod digitálního návrhu a softwarového vývoje systém na ipu

Guidelines:

Heterogenous computing systems in embedded devices are becoming increasingly complex and out of reach for small development teams, including academic development. Tools necessary to build performant, efficient systems have made little progress on many levels in decades, unlike software development tools, such as build systems and programming languages.
The task of this diploma thesis is to evaluate commercial, academic, and open-source tools and methodologies for the design and verification of systems on chip with a focus on systems implemented partially or fully on FPGA, to find and test opportunities for improvements in design accessibility, productivity, and quality. Furthermore, where feasible, to propose or implement enhancements and alternate solutions.

Bibliography / sources:

[1] G. D. Hachtel, F. Somenzi, Logic Synthesis and Verification Algorithms. Kluwer Academic Publishers, 2002
[2] C. Wolf, Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/

Name and workplace of master's thesis supervisor:

doc. Ing. Stanislav Vítek, Ph.D. Department of Radioelectronics FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **22.08.2022** Deadline for master's thesis submission: **10.01.2023**

Assignment valid until: **19.02.2024**

doc. Ing. Stanislav Vítek, Ph.D. Supervisor's signature	doc. Ing. Stanislav Vítek, Ph.D. Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt   Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to thank the diverse and mostly anonymous industry professionals who shared their first-hand experiences with me, including several developers of the open EDA tools cited. Industry folklore presented in this work as motivation is condensed from their helpful answers to my questions. I would also like to thank all academic workers who open their materials to the world, prof. Zvánovec for supervising and supporting my previous thesis topic, and my family.

Declaration

I hereby declare, that I have carried out the presented work individually, and that I have listed all used literature in compliance with the applicable laws and requirements.

Bc. Emil Jiří Tywoniak

Prague, 10th of January, 2023

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Bc. Emil Jiří Tywoniak

V Praze, 10. ledna 2023

Abstract

While the best practices in digital design are well covered by literature, the operating principles of the underlying tools is described only in scarce, deeply specialized publications, if at all. This thesis aims to explore the algorithmic problems and their state of the art solutions at the intersection of digital hardware and software with full context, and to provide a foundation for further inquiry. Its main focus is on the theory, merit, and accessible implementations of problems in logic synthesis, formal verification, and physical design automation. To demonstrate these software tools, open designs are presented throughout, including the CTU CAN FD IP core. In topic conclusions, a case is built for a fully open flow for education, research, and commercial development.

Keywords: logic synthesis, formal verification, physical design automation, placement, routing, EDA, HDL, FPGA, ASIC, Yosys, OpenROAD

Supervisor: Associate Prof. Ing. Stanislav Vítek, Ph.D.
Faculty of Electrical Engineering, CTU, Technická 2, 166 27 Prague 6 - Dejvice

Abstrakt

Standardní praktiky návrhu digitální logiky jsou formalizovány v dostupné literatuře, ale principy fungování potřebných softwarových nástrojů jsou popsány jen v omezených, silně specializovaných publikacích. Cílem této práce je zanalyzovat algorithmickou problematiku a jejich současná řešení v průniku digitálního hardwaru a softwaru v plném kontextu a poskytnout čtenáři základ pro další zkoumání a experimentaci. Práce je soustředěná na teorii, smysl a existující implementace problematiky syntézy logiky, formální verifikace a fyzickém návrhu integrovaných obvodů. Pro ukázky těchto softwarových nástrojů jsou použity otevřené nvárhy včetně CAN FD IP jádra ČVUT. V závěrech řešených témat jsou prezentovány argumenty pro plně open source metodologii pro vzdělávání, výzkum a komerční vývoj.

Klíčová slova: syntéza logiky, formální verifikace, automatizace fyzického návrhu, placement, routing, EDA, HDL, FPGA, ASIC, Yosys, OpenROAD

Překlad názvu: Analýza metod digitálního návrhu a softwarového vývoje systémů na čipu

Contents

1 Introduction 1

2 Design 3

2.1 Systems on chip . . . 3
2.2 Hardware description languages . 5
2.3 High level synthesis . . . 12

3 Verification 17

3.1 Introduction . . . 17
3.2 Functional verification . . . 18
3.3 Modeling synchronous circuits . . 19
3.4 Expressing properties . . . 20
3.5 Bounded model checking . . . 20
3.6 Temporal induction . . . 21
3.7 Proving liveness . . . 22
3.8 SMT solvers . . . 23
3.9 Theorem proof assistants . . . 26
3.10 Logical equivalence checking . . 26
3.11 Conclusion . . . 26

4 Synthesis 29

4.1 Combinational synthesis . . . 29
4.2 Sequential synthesis . . . 33
4.3 Physical synthesis . . . 35

5 Physical design automation 37

5.1 Synthesis backends . . . 37
5.2 Floorplanning . . . 39
5.3 Placing . . . 42
5.4 Clock tree synthesis . . . 45
5.5 Routing . . . 46
5.6 Static timing analysis . . . 52
5.7 Layout vs schematic checking . . 52
5.8 Circuit extraction . . . 53
5.9 Design rule checking . . . 53
5.10 Design for testability . . . 54
5.11 Physical design representation . 55
5.12 FPGA placement and routing . 56
5.13 Conclusion . . . 57

A Experiments 59

A.1 Yosys temporal logic counterexample loop detection . . . 59
A.2 GHDL PSL bug . . . 60
A.3 Amaranth combinational loop detection . . . 60
A.4 Minor Yosys bug fixes . . . 63
A.5 OpenROAD . . . 63

B Attachments 75

C Literature sources 77

D Bibliography 79

Figures

2.1 Unreachable combinational loop . 7

2.2 Compiling C with Clang and LLVM 9

3.1 Finite automaton with a reachable and unreachable unsafe state 23

3.2 Büchi automaton for “always eventually green”, adapted [56] . . . 23

3.3 Büchi product automata 24

3.4 Solver performance benchmark . 25

4.1 Boolean function representing data structures 31

5.1 OpenLane design flow [88] 38

5.2 Clustered graph adapted [92] . . . 41

5.3 Min-cut placement, adapted [92] 43

5.4 VCC and GND rails in a standard cell row 43

5.5 Four routing layers with alternating direction 47

5.6 L-RST separability, adapted [92] 48

5.7 Intel Arria 10 BEL, “Adaptive Logic Module”, adapted [134] 56

A.1 Cell size comparison, adapted [157] 64

A.2 CTU CAN FD IP core physical design 65

A.3 CTU CAN FD IP core physical design 65

A.4 CTU CAN FD IP core clock tree synthesis H-tree 66

A.5 TinyTapeout physical design, with `PL_BASIC_PLACEMENT` 68

A.6 TinyTapeout physical design . . . 69

A.7 TinyTapeout physical design . . . 70

A.8 TinyTapeout physical design . . . 70

A.9 TinyTapeout physical design . . . 71

A.10 Pipeline depth sweep 73

A.11 Pipeline width sweep 74

Tables

Chapter 1

Introduction

This work is the compilation of the results of my search for answers to a question I have asked myself years ago: what's the proper way to do digital SoC design? The answer is fragmented and scattered across several areas of active research and development, in very few locations, often behind closed doors. Many of the fragments are specific to software ecosystems of the big three (Synopsys, Cadence, Siemens) EDA vendors or tied to platform-specific tools. This motivated me to use open source tools to their maximum potential, which brought me to a new question: where do the limitations of open source tools come from? Are the obstacles to reaching quality parity or superiority to the closed ecosystems insurmountable? This work aims to demystify some areas relevant to these open-ended questions.

In Chapter 2, I present the ways of describing hardware in hardware description languages (HDL) in relation to software. Software programming language development has received a great amount of attention, with heaps of resources and diverse existing approaches. Meanwhile, HDLs that dominate the industry are considered canonical and untouchable, only somewhat extensible over time, while a market with software to cope with their limitations at scale thrives. I examine the levels of abstraction available to designers, and the varied ways alternative HDLs have been built.

Chapter 3 covers the theory and practice of proving digital design correctness and hunting for bugs, where open source tools greatly lower the cost of entry to building confidence in one's design.

In Chapter 4, I summarize the theory of logic minimization methods as implemented in ABC [1] and Yosys [2]. Yosys is a powerful open source logic synthesis toolchain, which allows user freedom in selecting optimization and conversion passes. It provides most notably a Verilog HDL frontend and netlist backends for interoperability with other tools. Yosys has been used internally in industrial EDA tools for small FPGAs [3], [4].

Chapter 5 explores the diverse algorithms needed to convert a synthesised design to integrated circuit geometry or an FPGA bitstream. Surprisingly, despite the recent chip availability problems and national security concerns, and the proposed funding for integrated circuit production capabilities, this

is not a field where Europe plays a significant role. However, open academic industrial-strength tools have surged in quality, integrated into the OpenROAD [5] project. I compare these tools to other open alternatives, and speculate about their industrial viability.

While smaller examples are presented for illustration of these chapters, I have used open designs to demonstrate and explore the available tools and to implement enhancements in Appendix A.

In this work, many general facts about the relevant problems and algorithms are introduced. In Appendix C I direct the reader to the very thorough but incomplete published literature and open course materials for further explanations of concepts not cited directly.

Analog and mixed-signal design represents a significant portion of the engineering effort of integrated systems, it is not within the scope of this work to delve into it. Automatic analog placement and design has been an open research topic for a long time, with many proposals and implementations, but its integration into real workflows remains low [6]. Beyond that, its nature provides few opportunities for inspiration by software development tooling, unlike digital design.

Chapter 2

Design

2.1 Systems on chip

Systems on Chip (SoCs) are complex integrated devices with general-purpose processors, memories, input and output cores that often implement communication protocols, and specialized coprocessors. In practice, an SoC designer integrates a large amount of separately developed IP blocks to implement each functionality.

An IP (intellectual property) block is a hardware module made to implement a specific set of functions conforming to a specification. Its functionality is defined by a set of hardware description source files. It can contain instances of other IP blocks, imported from elsewhere. Typically, to meet a wide set of requirements, it's parametrized with build-time configuration options. For example, an FFT IP core may be parametrized with size, internal bit widths and types, rounding mode, and many more [7]. Some parameters may be specific to FPGA (e.g. which elements to use as RAM) or ASIC only (e.g. scan chain insertion). Beyond hardware sources, there may also be synthesis constraints, and non-synthesizable descriptions like testbenches, simulation models, and memory map documentation. Hardware description sources may be obfuscated or encrypted to enforce copyright.

An SoC is simply an IP core with constraints to represent physical inputs and outputs, additional files for mapping IP core registers into memory, and synthesis/fabrication files. This makes IP core memory map documentation particularly interesting, since some IP is exposed to software running on one or more processor cores. When software is co-developed as hardware prototypes are built, the software should at all times have a correct map of the hardware. In the case of "bare metal" code (such as simple firmware or an early stage bootloader), this typically means generating C header files. Once an abstraction system like an operating system (OS) or real-time operating system (RTOS) is introduced, it's necessary to use whatever hardware information that system requires, typically, some form of a "device tree". For example, the Linux OS, Das U-Boot bootloader, and Zephyr

RTOS all have a very similarly structured device tree format. C headers are insufficient when an RTOS or OS use virtual memory, requiring a hardware driver to request resource access from the RTOS or OS. Device trees do not actually carry full register maps, only the starting address of a memory mapped device, so relative register maps still need to be created. Furthermore, a device tree fragment informs the kernel to load the correct driver, and can inform a generic hardware driver about the configuration or version of the hardware, so a single driver can support an entire product line of devices or cores.

SoCs implemented on an FPGA platform make use of its built-in “hard IP” blocks. These are stubs and configuration interfaces provided by the FPGA manufacturer to interface between developer-specified programmable logic and the fixed function logic in the processing system or in input/output blocks like serializes and deserializes (SERDES) or fully integrated communication protocol implementations like USB ULPI, PCIe, or Ethernet RGMII.

Even though SoCs contain numerous interfaces between hardware and software, which can be subject to experimental changes at many points in the development process, especially in the case of designs or prototypes targeting FPGA, exporting the interface to one block to be seen by another is often a manual process. Developing software and hardware in parallel is called hardware/software co-design. This requires interoperability between various design tools.

The IP-XACT (IEEE standard 1685-2014) [8] establishes a basic level of IP block interoperability between hardware design tools by carrying tool-agnostic information about the files required, as well as register memory map descriptions for software support. The Kactus2 graphical tool [9] uses IP-XACT to enable system-building capabilities similar to Xilinx Vitis and similar commercial tools, with no restrictions on the underlying synthesis and simulation tooling and target technology. For IP-XACT integration into custom project build systems, the ipyxact [10] Python-based parser can be used to extract IP metadata and generate C headers and memory map documentation and is used as a component of the Lattice Propel proprietary FPGA vendor EDA tool. LiteX [11] is an open HW/SW co-design project with its own build system for FPGA and ASIC SoCs, capable of instantiating feature-rich embedded Linux with a wide selection of processor cores and peripherals on many supported development boards. For IP core packaging and automatic creation of vendor tool project files, the FuseSoC [12] package manager integrates with the [13] project file generator. This setup is designed to kill the practice of downloading fixed version IP cores, and replacing it with a more software development inspired approach of taking locally made improvements to IP cores and “upstreaming” them to be shared with other users, who can easily upgrade to newer versions with fixed bugs.

Overall, the adoption of these open consolidated packaging and build tools remains low. Professional development teams naturally gravitate to whatever solution is most explicitly supported by their EDA software providers, while

open source developers, hobbyists, academics, and small commercial teams keep copy-and-pasting files by hand with builds being done on local machines instead of continuous integration that would guarantee build reproducibility. The "best practice" recommended by professionals is building in EDA tool batch mode directed by manually written TCL scripts, automated with GNU Make.

2.2 Hardware description languages

The goal of a hardware description language is to allow a user to express any working circuit in a concise, readable way, that can then be efficiently understood, processed, and converted by a synthesis tool, with the ultimate goal of simulation, FPGA programming, or integrated circuit fabrication.

As simulation and programmable logic allow a designer to evaluate their design choices at speed, it would be wasteful to work with a language that doesn't let the designer transparently influence the quality of the results. Also, it would be wasteful to force the designer to specify things that are arbitrary, repetitive, and solvable by an automatic tool. The conclusion from these two observations is clear: a good HDL spans multiple layers of abstraction.

In this chapter we will go through several HDLs, from standard to experimental. The majority of these, just like Verilog once was, don't provide a specification of their behavior that all implementations must adhere to, but only (often limited) documentation of the one existing implementation. This may disqualify them from formal design processes in engineering due to reliability concerns, but not from a feature evaluation.

2.2.1 Description architecture

There are multiple philosophies of describing the same circuit which come into play in different situations. It should be noted that in Verilog and VHDL, there is no way to restrict a module to only use one type of description—these are only conventions used as needed. However, in other languages, there may be limited support for writing on a given level.

Structural

Structural descriptions are close to hardware, in that they consider signals exclusively as outputs of simple modules with minimal usage of operators that need to be mapped onto available hardware primitives automatically in synthesis. There is no concept of a process or loop. This representation is close to a textual representation of a schematic. There is also some correspondence to the single static assignment (SSA) form of statements in compiler internal representations. Single assignment means that conditionally executed blocks are represented as join nodes in a well-formed graph with each assignment

an edge. Similarly, instead of if/then/else, structural hardware descriptions replace multiple conditional assignments with unconditional assignments to the output of a multiplexing operation. This limited expression power leads to the use of manually written structural descriptions to situations where a description needs to reliably map to available hardware elements, such as instantiating a physical resource on an FPGA or delay sensitive elements for a memory or programmable logic array implemented in silicon. This leaves no guesswork and no optimization opportunities to the synthesis tool. Automatically generated structural descriptions are much more common, since structural Verilog is commonly the interchange format between EDA tools. This specifically leads to bugs when multiple tools use different frontends due to the chaotic nature of Verilog's specification.

Dataflow

Dataflow architectures loosen the definition of structural architectures by allowing arithmetical and bit operators, not just modules that implement them. It can be argued that dataflow is isomorphic functional programming. In fact, Clash, a Haskell-based HDL, continues on with conventions from Haskell, where `let` blocks are discouraged in favor of pure functions or cleaner constructions like state monads.

Behavioral

Behavioral descriptions correspond to procedural programming.

Blocking assignments in Verilog are "executed" immediately and influence all subsequent assignments. This is often the natural way one would verbally describe what a module full of conditions does and saves time from forcing the programmer to rethink this description.

This can in some cases lead to unfortunate errors. A programmer looking at part of a module in isolation can miss a later assignment that overrides assignments seen so far, or leave a signal unassigned. For example, Verilog's signal model lets `reg` signals retain their previous value if not assigned. Consider the following reduced example, taken from a floating point arithmetical unit [14], in lst. 1.

On its own, `o_exponent = o_exponent + 1;` would create a logic loop, as shown in yosys `show` command output in fig. 2.1. But the code works fine in simulation. No matter the value of `a_exponent` and `b_exponent`, one of the first three branch conditions evaluate to true, and set `o_exponent`. Then the incrementing statement alters it by one. However, some tools might see that if none of the three conditions hold, there is a combinational loop, after the required yosys `clk2fflogic` pass, which replaces the D-latch gating the combinational loop with a buffer (BUF). It would be theoretically possible to check if such an assignment is always shadowed by preceding ones, but

```
always @ ( * ) begin
    if (a_exponent == b_exponent) begin
        o_exponent = a_exponent;
    end else if (a_exponent > b_exponent) begin
        o_exponent = a_exponent;
    end else if (a_exponent < b_exponent) begin
        o_exponent = b_exponent;
    end
    if(o_mantissa[24] == 1) begin
        o_exponent = o_exponent + 1;
    end
end
```

Listing 1: Behavioral Verilog generating a combinational loop

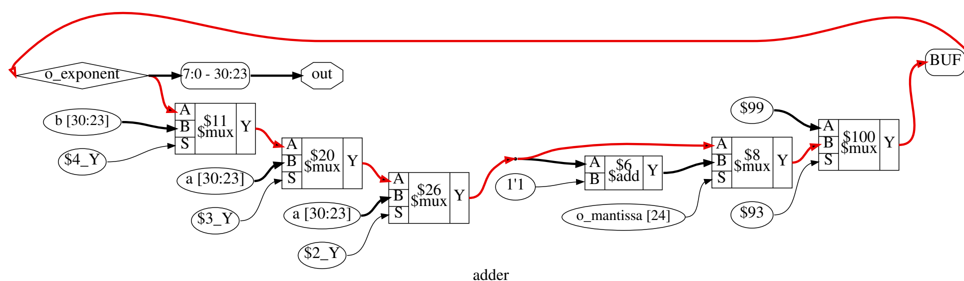


Figure 2.1: Unreachable combinational loop

this would actually in the general case require checking satisfiability on the condition expressions with a solver. However, it's easily rectified by changing the last `else if` to `else`, since the developer knows the intent of these conditions.

This demonstrates an easy programmer error that is enabled by Verilog's design. Amaranth takes a different approach: assignments in the combinational domain override previous ones, but don't use the values of previous assignments to the signal being assigned to. This means that the equivalent of `o_exponent = o_exponent + 1;` will always create a combinational logic loop, since previous assignments are "shadowed" by this one, like variable declarations in C, or any definitions in Haskell. The Amaranth documentation forbids combinational loops. At the time of writing, Amaranth doesn't detect logic loops, but it's a planned feature. I present my proposal for its implementation in Chapter A.3.

High level synthesis, discussed in Chapter 2.3, can be considered an advanced form of behavioral architecture description.

In languages like VHDL, modules can have multiple implementations to support varied targets, for example, simulation and FPGA. This is a powerful idea, allowing productive composable definitions of hardware on a fine-grained level allowing for hand-optimized circuitry as well as a readable, easy to

simulate, abstract level. However, like any functionality duplication, it creates significant risks of introducing bugs in their inconsistencies. Formal methods may alleviate this, see Chapter 3.10.

2.2.2 Implementation approaches

Computer languages have similarities in their processing pipelines. As an example, the compilation pipeline from C source to machine code is illustrated in fig. 2.2. To generalize, such conversions always require a "front end", consuming the typically text-based input, some processing, and a "back end", emitting the desired target artifact. This applies not just to programming languages, but even to things such as Pandoc. Its "Markdown reader" front end reads this text, builds an abstract syntax tree (AST) out of it, expands Markdown features into a LaTeX file, and runs pdflatex to convert it into the PDF, which you are reading now. Additional processing steps can be added in between as "filters" that alter the AST.

In the case of LLVM, cleanly separating the front ends (like Clang), shared optimizations, and back ends has allowed compiler developers to easily add support for new languages and new computer architectures while collaborating on shared optimizations.

Some degree of this approach exists in commercial tools. Since moving code between various synthesis, simulation, and verification systems is desirable, software tools by Synopsys and Cadence have outsourced their Verilog front ends to a third party, Verific Design Automation. It's a common story that language support developed in-house inevitably leads to great pain and inconsistency, exacerbated by Verilog's often ambiguous or hard to implement specification and lack of care for VHDL support. As far as commercial backends go, there isn't much variety. Their outputs are typically undocumented proprietary formats—since FPGA bitstream formats reveal information about the architecture, their format is never published by vendors, and published reverse engineering outputs are considered violations of intellectual property. On the other hand, ASIC fabrication output formats are open standards.

Within the realm of modern and experimental HDL, tooling has been developed in notable different ways.

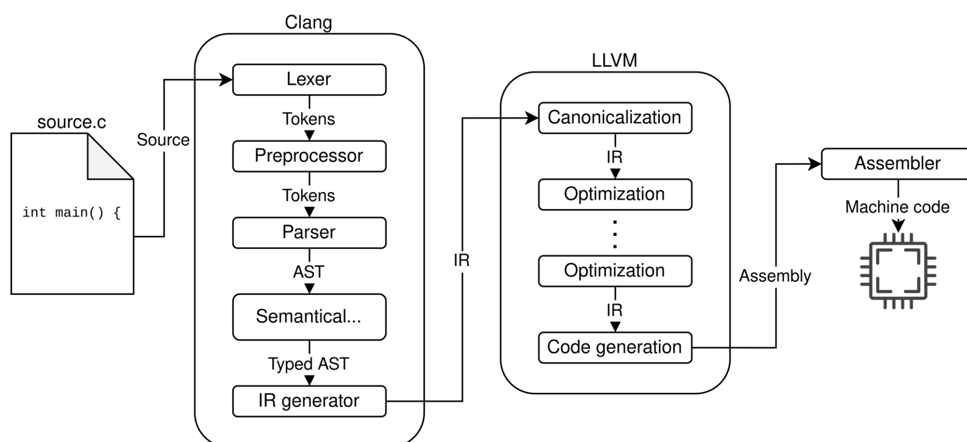


Figure 2.2: Compiling C with Clang and LLVM

2.2.3 Front end

The conventional way to design a language involves formally specifying its grammar, typically in the Backus-Naur form, or an extended version of it, and to then use efficient lexer and parser generators to transform the source to an AST. Some manual code is often needed to handle string literals and correct error reporting.

However, this demonstrates an opportunity to use existing infrastructure for a programming language. In fact, it's possible to extend a programming language to an HDL without modifying its implementation, as an embedded domain-specific language (eDSL). They are often implemented as libraries and define their own types and operations. Their syntax can be limited by the potential of the host language to be extended. For example, embedding a language in C forces the programmer to build it using the C macro system, while embedding it in Lisp or Rust brings much more reliably usable syntactic macros. Furthermore, DSL-based languages have lower costs to implementing systems for HW/SW co-design since it's possible to write software with full knowledge of the hardware interfaces without needing to extract and convert hardware metadata, or at least brings down the barrier to developing tools that do this conversion.

It should be noted that an HDL that looks and feels very similar to a programming language doesn't imply it's implemented as en eDSL. VHDL may be inspired by Ada and Verilog may be inspired by C, but tools for neither are necessarily even implemented in those languages.

2.2.4 Back end

The final output of an HDL's implementation is typically structural Verilog or an intermediate representation (IR). Language and synthesis tools may have serializable IRs, which can be used externally for simulation, language support,

and debugging. As an example, RTLIL is a part of Yosys [2], and FIRRTL [15] a part of the Chisel [16] language. Both are focused on low-level circuit representation. Proprietary ecosystems have IRs that aren't documented publicly and can only be supported in partnership with the ecosystem owners. Some cross-language, cross-tooling IRs have been proposed, most notably, LLVM MLIR [17] provides several "dialects" for targets such as GPUs and accelerators, including a "HW" dialect, developed by the LLVM CIRCT project [18]. LLHD [19] is a proposed hardware-specific IR with an interesting event model scheduling structure, now a part of CIRCT. Cayx [20] is an IR focused on representing high level synthesis programs for elaboration into RTL or an RTL-level IR.

2.2.5 Existing languages

The lingua franca of digital design is Verilog [21], a language originally designed specifically as a simulator frontend language at a time when simulation wasn't a basic practice. Its specification contains legacy features that are impractical and typically unsupported, such as signal drive strength levels. Productivity in this language is lacking for real-world projects beyond small modules. Coping with the lack of metaprogramming for example requires SoC designers to use or build external tools that generate Verilog.

To remedy this, SystemVerilog [22] was designed to provide much of what was missing. While Verilog does support `generate` constructs, allowing for conditional and iterative hardware building, it doesn't allow the user to pass build-time parameters to modules. This means inherently parametric designs such as a FIR filter are not parametrized at the point of instantiation, but in project files for an external tool. Multi-signal interfaces between modules are missing as well, so implementing buses such as Wishbone requires the same 9 signals copy-and-pasted to every block with this interface. SystemVerilog adds interface declarations with `modport` constructs to define directions for each role of module on the interface, for example, a stream bus master sees data as an interface input, while a slave sees it as an output. Even though SystemVerilog improves on Verilog's type system, it also inherits from Verilog a lack of many productive features. For example, there is no notion of a clock or clock domain. A clock is a regular signal with no safety checks, even though it's risky to treat it as such in design. Clock domains are implemented by manually assigning the clock of a module to its submodule's clock input, adding boilerplate, delegating any checks of the sanity of the clock domain hierarchy to external tools. Also, SystemVerilog is inspired by the C programming language to its detriment, with include declarations rather than a module-like system to increase a file's scope to declarations in other files, as well as leaving the user to rely on macros to reduce boilerplate in interface connections. Assignments are also rather unrestricted, presenting risks of creating glitches and unwanted latches in designs, as shown in Chapter 2.2.1. SystemVerilog also brings a plethora of strong features with varying levels of popularity and support, such as an object-oriented programming paradigm

with inheritance, and powerful assertion constructs. It is common for open source SystemVerilog implementations to silently skip implementing many of its features. A notable open implementation is the Slang [23] front-end, based on LLVM Clang, which fares well in independent support tests [24].

Transaction-Level Verilog (TL-Verilog) [25] adds greatly expressive features for common design kinds such as pipelines and bus transactions, leading to more succinct, highly parametric hierarchical code than SystemVerilog, almost reaching HLS-level abstraction (see Chapter 2.3). It also adds some safety and consistency by automating away assertion boilerplate on interfaces. Silice [26] has similar design goals, with a focus on supporting instantiating hard IP blocks in FPGA.

VHDL [27] provides a level of productivity similar to SystemVerilog with stronger typing, requiring explicit, well-defined type conversions. Even though it was designed more deliberately as a design language, it has superior simulation semantics, reducing the opportunity for simulation inconsistencies across implementations [28]. Sadly, its support level is lower in both commercial and open source tools than Verilog. GHDL [29] is an open but incomplete implementation of a VHDL front-end and simulator.

The inconsistencies and inefficiencies inherent to these languages have motivated a great number of experimental languages with varied design goals and implementation approaches.

Chisel [16] and Spinal [30] are embedded in the Scala programming language. While Scala offers great language embedding support, leading to metaprogramming beyond SystemVerilog's `generate` and simple parameters, it can lead to convoluted errors, revealing the abstract object-oriented internals. Both support inheriting clock domains, but only Spinal detects illegal clock domain crossing. Both languages integrate the Verilator simulator, which transpiles Verilog designs to C++ for fast simulation [31]. Both also provide formal verification support via Yosys.

Clash [32] is embedded in Haskell, making use of its powerful type system. However, I have observed the overhead of writing hardware in Haskell to lead to a programmer experienced in Haskell and a clear idea of the design being built to spend much more time thinking about the type system rather than the hardware being built. Bluespec [33] is a Haskell-like HLS language more focused on practical design and bringing the "atomic transaction" philosophy from the field of concurrent system design, which fits digital hardware well. Bluespec Verilog combines these semantics with a SystemVerilog-like syntax.

Amaranth is embedded in Python using operator overloading to express custom operations, and context managers to construct control flow elements. Amaranth provides a reasonable level of safety without sacrificing much expressive power. Modules with synchronous logic inherit clock domains, but clock signals can still be accessed. Assignments are inherent to a domain, and can't be mixed. The difference between a hardware signal and a build-time Python variable is fairly clear. My experience with Amaranth has shown

that while flexible and safe, the amount of Pythonic boilerplate it imposes is discouraging, parts of designs silently go missing if the assignment to a combinational domain is forgotten, and interfaces are currently impractical, although modules can easily inherit from a class with signaling boilerplate for an interface. Amaranth targets the Yosys RTLIL IR, optionally generating Verilog via Yosys, and includes great constructs for supporting development boards or custom hardware as well as automating proprietary build tools. It also inherits VHDL's simulation semantics and implements a Python-based simulator as well as the CXXRTL simulator which is similar to Verilator.

Experimental proof assistant-based hardware design languages Kôika [34] and Kami [35] are inspired by BlueSpec and built as a library for Coq [36]. While evaluating their is difficult due to the nature of programming in Coq, their scope is unique: hardware designed in Coq is meant to be a proof of its own correctness.

While limited formal comparisons between alternative HDLs have been published [37], it's hard to know beforehand if a language is sufficient for one's design goals. Although hardware description isn't *the* problem with hardware design quality or accessibility, it certainly is *a* problem, and diverse solution approaches have been presented in this section.

2.3 High level synthesis

High level synthesis (HLS) is a general name for a diverse set of languages and tools for hardware development. The goals of these tools are typically

- Ease of use for software developers by using similar language constructs to programming languages
- Direct conversion of computer programs to hardware that is logically equivalent
- Single language, single repository hardware software co-design
- Testing and verification on a system level without creating abstract models such as SystemC
- Single definitions for interfaces between hardware and software
- Given fixed hardware and software interfaces, automatically determine what sections of unannotated programs to offload into dedicated hardware accelerators

The review in Hempel (2019) [38] provides a thorough analysis of the design of existing high level synthesis languages.

There are many challenges to abstracting hardware away. Most commonly, the design space is severely constrained by the abstractions and interfaces chosen by the tool, as the designer loses a lot of direct control. This is generally desirable, as a good tool can do a better job than a good designer at certain tasks.

To illustrate this point, let us review a humble C compiler's structure as illustrated in fig. 2.2. First, lexical analysis is performed, which processes source code into a set of tokens, such as type, variable, assignment operator, or opening bracket. The code for this stage is called a lexer and its grammar is specified by regular expressions. From these, the parser builds an abstract syntax tree (AST), according to syntax rules, defined as a formal grammar where terminals are lexical tokens. Semantic analysis then assigns types to expressions and checks if the tree "makes sense"—for example, if methods are called with correct argument types. At this point, we would be ready to emit some sort of assembly or machine code. However, if code was generated directly from the AST, targeting various computer architectures would require a large degree of redundant code, as they tend to share many characteristics. Also, the code would be completely unoptimized, and optimizing emitted code is memory intensive and slow. For this reason, a generic intermediate representation (IR) is introduced to capture all language constructs. Optimizations then are done on the IR. An IR can be further turned into one that more closely matches the available instructions by "lowering" it into another IR with finer operation granularity.

For register based instruction sets, local variables and their temporary products need to be placed into registers to be operated upon by for almost all instructions to be able to operate upon them. The C programmer isn't in control of when and where variables in their code will be allocated. It would be time-consuming, error-prone, and would lead to slower, unportable code, if register allocation was specified manually. When it's necessary, for example in operating system implementations, assembly language is used instead.

The compiler instead does register allocation automatically. It allocates variables to virtual registers, and then maps them to physical ones matching the target architecture. One conventional way to perform this is to build an interference graph, where two virtual registers are connected ("interfering") if their variables should be at some point in time both "alive". In short, a variable is alive at all points in the program runtime between its first write and last read before the next write. These graph nodes must then be "colored" by as many colors as there are available physical registers, such that no two interfering registers have the same color. This is a modified instance of the graph coloring problem, where many rarely used virtual registers may be stored on the stack and accessed into one dedicated register as needed. This is called "spilling". As accesses to memory are orders of magnitude slower than register accesses, this must be minimized. The graph coloring problem is NP-hard [39], and even though well-formulated intermediate representations lead to optimal register assignment being possible in polynomial time, adding optimizations to remove redundant register moves brings the complexity back to NP-hard [40]. As a result, heuristic approaches, SAT solvers [41], or deep learning [42] are used to get decent results in reasonable time, offering superior results to what a programmer might write by hand.

Similarly, a compiler may analyze several loop over an array, or more easily,

functional declarative iteration methods such as map, filter and reduce, and not only merge them into one pass ("polyhedral optimization [43]"), but also process multiple elements in one iteration. This can make use of vector processing capabilities of a given processor ("automatic vectorization [44]"). The combination of these techniques often outperforms any manual attempts by a skilled programmer, aware of the vector capabilities of their processor [45].

While these techniques are automatic, the programmer can set or limit their parameters for a given source file or through adding compiler specific annotations in the code, for example for a single loop. Let us now consider similar techniques allowed by program behavior abstraction, but in the world of digital logic.

2.3.1 Pipeline scheduling

Quality of a correct logic design is multifaceted. Most commonly, developers and software tools optimize the design for power, performance, and area (PPA). A design can be optimized for a trade-off between these, or strongly biased towards one or another target.

For example, in the case of a RISC processor, a single-cycle implementation is quite compact, but has a long critical path. A critical path in a synchronous design is the signal path from a flip-flop to another, on which the signal propagation delays add up to the greatest overall delay. The maximum frequency at which a design can run is inverse to the critical path delay. If the design was clocked at a higher frequency, signals would not be consistently settled when sampled by a flip-flop, possibly propagating metastable signals, leading to undesirable behavior, not described by the design.

When higher throughput is desired, a pipelined implementation can be used. All paths are now broken up with multiple registers, breaking up the design into multiple stages. With N stages, we can achieve a maximum speedup of N, as long as instructions don't depend on instructions, that haven't yet propagated through the pipeline and committed their results to register or memory. This requires careful testing, since a data or control dependency arises, a "stall" state must be propagated through the pipeline, until the results are available.

Data flow designs that do not have this kind of loop are called feed forward. This presents an opportunity for automatic pipelining, also called pipeline scheduling. Common use cases include data processing for a feed of audio or video, or a part of some more general purpose accelerator. In this case, there is little advantage to forcing the developer to decide and manually tune which paths to cut where while paying attention to preserving correctness, avoiding cross-contaminating the stages.

To automatically split a feed forward design into stages, we need to know how long the critical paths are. For that, we need to model delays of the elements

they are built out of. Tools such as Google XLS [46] (designed for ASIC) take data from the PDK (process development kit) for their fabrication node, evaluate a set of elementary arithmetical and logical operations for a number of input counts and bit widths. This thorough but limited data is used to fit parameters for a simple multivariate delay model. This model then gives an approximate delay for any given element.

Chapter 3

Verification

3.1 Introduction

Complex digital logic designs such as systems on chip are almost universally designed using a "globally asynchronous, locally synchronous" architecture. As the size of the design becomes significant, it is desirable to verify only each element of a synchronous domain in isolation, and if proper faith in the final system is desired, to use then those proven properties as assumptions in verifying the final asynchronous design. For example, if we prove that a memory module will always respond to a read request of up to length N within k cycles, we can use that as a simple model of the memory module to model the entire system out of such well-behaved "black boxes" and evaluate system performance without modeling unnecessary details. A property is a well-defined characteristic of a design, informally describable as "something bad may never happen". Expressing properties is referred to as "asserting" them. In the context of digital hardware, "something bad" is some combination of partially described signal values. We describe this in some sort of formal logic, as described below. By "never happen", we mean "there is no set of input, that would result in the property being true at any point in time". This naturally gives rise to a categorization of verification methods, some of which are fundamentally weaker.

- "No set of inputs" may be too hard to prove: while formal verification actually proves that there is no set of inputs that triggers a bug, functional verification in general only checks that the design responds correctly to some input. This is the extent to which verification is often done (and taught). While in practice it almost guarantees that even in simple designs there is some case the designer didn't take into consideration where bugs can occur, it's still a valid way of catching bugs early. Functional verification is performed by feeding a module with manually written inputs. Testbench is an HDL module or program that instantiates or interfaces with a module, feeds such inputs into it, and checks safety properties on the resulting outputs.

- "Never" may be too hard to prove: bounded formal verification methods only find bugs for the first k cycles of a design's life, while unbounded methods prove bugs are actually impossible.

"No set of inputs" can also include inputs that are expected to be guaranteed by whatever module or program is interfacing with the design under test. For example, if a programmer is instructed to write to the registers of a microcontroller peripheral in a certain sequence, it might be acceptable to have bugs reachable when that sequence is performed incorrectly. In that case to eliminate these false positive failing assertions by "assuming" a property concerning these inputs. Bounded formal methods become equivalent to unbounded if the design only has k reachable states or fewer. However, the state space of many designs is too large to make that a possibility. A design containing N flip-flops (or N register bits) will have 2^N states in total. If it does any sort of arithmetic, impractically many of these states will be reachable.

Execution time of automatic verification methods is always an important consideration. Ideally, the engineer writes assertions while designing, makes the design work, and no more work is necessary. In practice, it's common for small modifications done later to cause regressions, i.e. for assertions passing on previous design revisions to fail. This must be checked as early as possible. Similarly to software engineering, this typically takes the form of continuous integration servers building the design and re-verifying everything, detecting regressions. This requires a design's full verification suite to be possible to run within hours. This even influences the way assertions are written—sometimes more assertions can rule out false positives with up to exponentially less work.

3.2 Functional verification

3.2.1 Mutation testing

Since testbenches are the most common way of demonstrating that a module is capable of working, several methods were devised to make them build more faith in the user in their correctness. First, it should be noted that this is also almost universally how software is commercially developed. A developer writes for each minimal functional unit of a program (typically a function) a "unit test" with fixed inputs and outputs. Additional tests can then run on the entire program composed of these units. Since software correctness is in general pretty much impossible to prove automatically, coping with unit tests' limitations has lead to some clever techniques. One of them is coverage analysis, which can be extended to mutation coverage analysis. Let us first consider what good tests look like. Good tests pass whenever their units under test behave as desired, and fail when they don't. This suggests the following:

1. Some test must fail, if the unit is modified in any meaningful extent. Otherwise, some part of the unit is untested and can't be trusted.
2. Every test must be able to fail, if the unit is modified in some way. Otherwise, this test is worse than useless, since it creates a false sense of program correctness.

A weaker version of 1) can be done by analyzing test coverage, that is, the set of program features that have been explored when the test was run. Features can mean functions called, statements (reported as lines covered), edges in the program's control flow graph, or control flow operator conditions being evaluated to true at some point and false at another. The tooling for coverage of programs written in procedural programming languages requires attaching a debugger and controlling it as well as analyzing its findings with knowledge from within the compiler.

Writing testbenches as fully specified input sequences (directed testing) is a time-consuming practice. Generating inputs as completely random will result in most of the inputs triggering functionally the same behavior cases, while some cases stay ignored. However, it's possible to partially specify the sequences as "constrained random verification", in which case the developer tunes the constraints to reach as many distinct modes of function or behavior cases, and uses coverage information as feedback.

In the case of hardware functional verification, while hardware corresponding to a higher level model can be checked in simulation similarly for coverage of the higher level source, it isn't a common technique. More commonly, explicit coverage statements written similarly to assertions in the HDL source are checked. This typically is used to verify that a test successfully triggers some feature of the hardware, as signified by some internal signal value sequence.

Checking both 1) and 2) requires restricting the sorts of modifications we can introduce into the artifact, meaning program or circuit to keep the procedure relatively simple. In software, operators can be replaced with others, statements can be erased or replaced with constants. In hardware, similarly, signals can be set to constants or operations on other signals. The reader may notice that this resembles testing silicon for faults with a stuck-at model.

An additional benefit of coverage analysis is that not just syntactically but also potentially functionally dead code can be found.

3.3 Modeling synchronous circuits

As opposed to testbench functional verification, which operates directly on the description on the circuit, model checking [47] operates on a model of such a circuit. Any correctly built (deterministic, uninfluenced by external) synchronous digital logic circuit with finite memory is a finite state machine (FSM). In design, Mealy and Moore machines are often differentiated. How-

ever, this is unimportant for synchronous designs. A Moore machine can be easily converted to a Moore machine by moving labels from states to all their incoming edges. The extra expressive power of a Mealy machine comes from its ability to respond immediately to inputs, corresponding to a combinational path from an input to an output. As we're interested particularly in synchronous modules within systems on chip, it is typical that all their outputs are registered, so Moore is sufficient. If we were interested in converting an arbitrary Mealy machine to Moore, we would need to label states with their incoming edges, creating a machine with outputs delayed by one cycle. This requires each state to have several copies, up to the size of the output alphabet, and creating an undefined or arbitrary output in the first cycle of the Moore machine's run.

Asynchronous designs in general have to be modeled with Petri nets [48], a technique known from concurrent system modeling. Asynchronous design is outside of the scope of this work.

Now we have a synchronous circuit model which can be simulated, but we're interested in checking its correctness. From signal values and assertions, we can build atomic propositions, and label states with boolean vectors with bit i set if and only if the atomic proposition i holds in the given state.

3.4 Expressing properties

"Invariant" is a property declared to hold at every step and composed of a first order logic formula built out of single atomic propositions. It has no notion of time.

"Safety properties" are sequences of formulas recognizable by a finite automaton. Invariants are a special case of safety properties.

"Liveness properties" (assertions of the form "eventually, something happens") require us to widen our scope to temporal logic [49], which allows us to describe properties of infinite words. For synchronous hardware, linear temporal logic (LTL) is sufficient. For concurrent and asynchronous designs, "branching" paths have to be considered, which is implemented in computation tree logic (CTL [50], CTL* [51]). Safety properties are a special case of liveness properties.

3.5 Bounded model checking

With bounded model checking (BMC), we prove the following: "For the first N cycles, there is no input, that would trigger a bug". This isn't a method to guarantee circuit correctness, but to find bugs. When state has M bits, and N is greater than 2^M, then BMC proves correctness, because it traverses all states. However, the state space is typically too large for this approach.

When dealing with purely boolean variables, the minimum k for which bounded model checking can prove any linear temporal logic formula has an upper bound of 2^d where d is the diameter of the model state graph [52]. When this is achieved, the method becomes unbounded. As the boolean type is insufficient for many kinds of digital circuits, we can consider atomic formulae composed of non-boolean variables to be extra hard-to-evaluate boolean variables. This aligns with how CDCL(T) SMT solvers work (see Chapter 3.8).

3.6 Temporal induction

Temporal induction allows us to actually prove safety properties [53]. It follows a simple scheme:

- Prove BMC of depth k as the base case. If it's disproven, we have a valid counterexample
- As the inductive step, prove that given k successive states where asserts hold, $k+1$ always holds too

If the inductive step fails, there are three options:

1. Induction starts from a reachable state, and the assertions are proven not to hold
2. Induction starts from a safe but unreachable state, and increasing the k will remove this false positive
3. Induction starts from a safe but unreachable state, but contains a loop, so increasing k won't remove this false positive

The k sufficient to prove the property isn't practically knowable. It's equal to the longest path in safe unreachable states, and it's impractical to construct the state space. Also, in particular, counters cause loops in control logic to appear as meaningful changes, greatly increasing the required k. Using temporal induction on designs with counters requires carefully disabling the counters with a signal asserted in testing.

There are multiple ways of detecting case 3.

- Detect loops in counterexamples
- Manually break loops, asserting some of their states to be unsafe
- Automatically break loops with “path constraints” [52]

For example, trying to prove the code in lst. 2 with a depth of 20 will lead to a loop in unreachable states, with the counter variable starting off somewhere close to 200, and up- and down-counting to it

```
module demo(input clk, mode, output reg [7:0] cnt);
  always @(posedge clk) begin
    if (cnt != 0 && mode == 0)
      cnt <= cnt - 1;

    if (cnt != 99 && mode == 1)
      cnt <= cnt + 1;

  end
    initial assume (cnt == 5);
    assert property (cnt != 200);
endmodule
```

Listing 2: Correct assertion example

As a simplified illustration, the counterexample caused by the failing inductive step is similar to a path in fig. 3.1, looping arbitrarily long in the bottom two safe unreachable states, until it enters the unsafe unrechable state.

Constraining the module in lst. 2 harder with `assert property (cnt < 100);` will make all states used to disprove the previous assertion invalid. All counterexample traces would now have to start with `cnt < 100` and can't increment beyond 99 because of the `cnt != 99` condition.

3.7 Proving liveness

Since liveness properties are statements about infinite words, we need to introduce a new approach. To check whether sometimes happens infinitely often, we introduce a finite automaton, which accepts an infinite path, if and only if it that "something" happens infinitely often. A (non-deterministic) Büchi automaton (NBA) is an automaton which accepts a path if it infinitely often passes through at least one accepting state. We can create one that corresponds to a liveness property, like "the semaphore will always eventually turn green" in fig. 3.2. To prove this property, we can now take an FSM model of the semaphore, labeled with atomic propositions like "green" and "orange" and "red", and build a new automaton through a modified product construction. This automaton will have states that are the Cartesian product of the sets of states in the system model and assertion automaton, but the automaton consumes as inputs the labels of the model automaton's states. Also, its initial state will be the tuple of the model's initial state and the assertion automaton state reached when it consumes the label of the model's initial state. If we can find a path with a prefix and a loop that doesn't visit the accepting state, then this path is a counterexample; otherwise, the assertion is proven. If we only construct reachable states of the product automaton, and find strongly connected components (SCC), we can in polynomial time

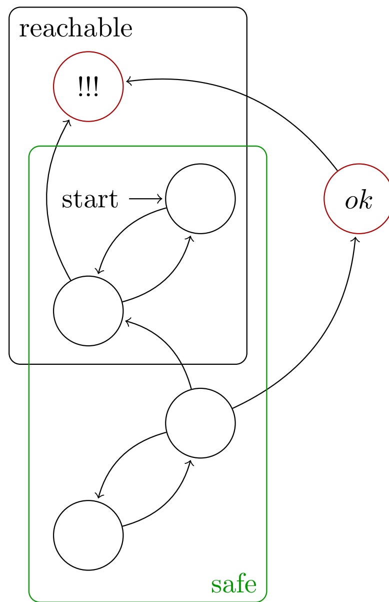


Figure 3.1: Finite automaton with a reachable and unreachable unsafe state

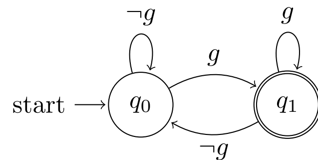


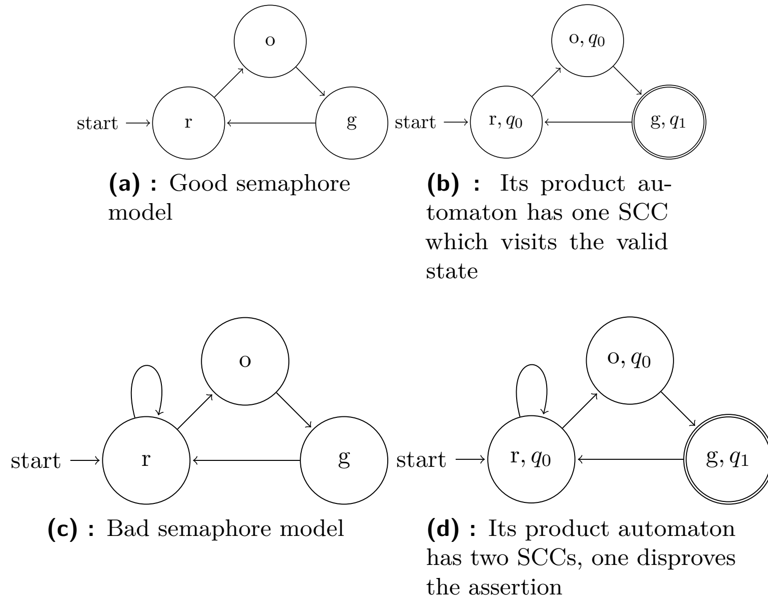
Figure 3.2: Büchi automaton for "always eventually green", adapted [56]

[54] find or disprove the existence of such a loop. If we found the loop, we can find a path to it with a simple linear-time search. This provides a counterexample. However, constructing the state space is exponential, and LTL property checking is PSPACE-complete [55]. The state space can be explored as it's being constructed, so counterexamples may be found quickly.

3.8 SMT solvers

Automating BMC and temporal induction requires tools to check the consistency of logical propositions automatically. Initially, binary decision diagrams [57] (as described in Chapter 4.1.3) were used to compress the state space being explored, but still were building the state space. To handle reasonably large designs, "implicit-state" tools are required.

In practice, SMT (satisfiability modulo theory) solvers [56] are used, as described below. To model assertions about state machines, we need to model the state machine in the language of formal logic. This can be implemented by declaring a state variable and defining its transition relation as well as the initial state. Then, we can convert assertions to satisfiability problems that



(a) : Good semaphore model

(b) : Its product automaton has one SCC which visits the valid state

(c) : Bad semaphore model

(d) : Its product automaton has two SCCs, one disproves the assertion

Figure 3.3: Büchi product automata

implement BMC or temporal induction by "unrolling" formulas about states over time.

The Boolean satisfiability (SAT) problem is the problem of determining the existence of a variable assignment for the Boolean variables a Boolean function is constructed out of, such that it evaluates to true. Typically, a restriction is placed on the form (representation) of the function, and often such a vector is meant to be constructed, not just claimed to exist. Conjunctive normal form SAT is one of the archetypal NP-complete problems. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [58] is a basic automatic SAT solver scheme, which searches the possible variable assignments and makes use of opportunities to simplify the problem as it partially specifies the assignments and backtracks if the simpler constrained problem is unsatisfiable. The available simplification rules are the following:

- Basic simplifications, which apply on formulas with literals, such as $\neg\top \rightsquigarrow \bot$, or $\top \vee \ldots \rightsquigarrow \top$
- Unit propagation, which assigns values to single clause literals: $[Q \vee R] \wedge \neg Q \rightsquigarrow [\bot \vee R]$
- Pure literal elimination, which assigns values that always appear either only negated or only not negated: $[\neg Q \vee R] \wedge S \rightsquigarrow [\top \vee R] \wedge S$

In a heuristic tree search fashion, unless the formula simplifies to a literal, a variable and the literal to set it to is chosen according to a heuristic, and the more constrained formula is recursed with.

SMT solvers extend SAT solvers with support for formulas on non-boolean types, for example, arithmetic over integers, bit vectors, or arrays. The DPLL

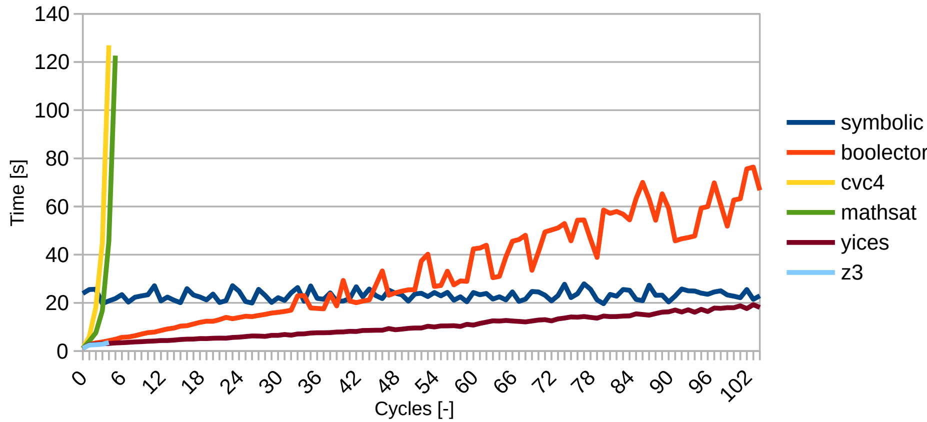


Figure 3.4: Solver performance benchmark

approach is insufficient. To illustrate, consider the formula $(a > 1000) \wedge (a = 30b)$. If DPLL was meant to be used, various values for a and b would be set and backtracked out of, taking a large amount of cycles (unbounded for arbitrarily large integers), and ignoring the (to a human) obvious constraints the formula sets for their values. Adding some level of reasoning ("T-solvers") about these additional operations like integer addition is called DPLL(T) or CDCL(T). CDCL, meaning "conflict driven clause learning", defers solving problems in the added non-Boolean theories to when it really matters for the overall satisfiability, and finding conflicts between theory solver results. These need to integrate tightly to a SAT solver. As an alternative, for some theories, it's possible to convert every operation into a Boolean formula on the individual bits of a binary representation of the types. This is called "eager" or "bit-blasting". In such implementations, the SAT solver used after this conversion can be chosen freely by the user. Neither approach seems to be obviously better for the theories necessary for typical hardware design verification, quantifer-free bit vectors ("QF_BV"). The performance of various solvers can vary greatly theory to theory, problem to problem, circuit to circuit. For example, the IsaSAT solver won the 2021 EDA Challenge [59] while coming last in the SAT Competition 2022's main track [60]. It can be also be seen in fig. 3.4, which compares several CDCL solvers used with Yosys. This benchmark is taken from a paper presenting rtlv [61], a symbolic evaluation tool, applied to the problem of verifying an SoC deterministically clearing its flip flop state in a set number of clock cycles after asserting a reset. The SMT solvers used are Boolector [62], [63], CVC4 [64], MathSAT [65], Yices [66], and Z3 [67]. Because of these performance differences, the open-sources Symbiyosys [68] verification automation tool based on Yosys provides the `--autotune` option, which compares and reports solver performance for a given verification task. The z3 solver isn't plotted fully because it crashed after a few iterations.

3.9 Theorem proof assistants

Automated theorem proving software like Coq [36] and Lean [69] provide a way of verifying proofs. While they have been used to prove hardware design correctness [70], their usage requires a great amount of manual effort. Their principle of operation is to provide a programming environment for writing proofs guaranteed to be true, with some automation functionality with "tactics". Meanwhile, SMT solvers are just clever exhaustive search algorithms, but require less user interaction. Also, modifying the specification or design may require rewriting proofs.

3.10 Logical equivalence checking

Automatic optimization steps in synthesis or manual optimizations can lead to bugs in the described behavior of a circuit. To detect these, logical equivalence checking (LEC) can be used. One way of implementing the equivalence of two circuit implementations is to connect their inputs pairwise and to XOR their outputs also pairwise. This new output vector then gets OR'd together. This constructs a "miter" circuit. The miter circuit will have its output set if the two circuit implementations have any differences in their outputs given a set of inputs. Now, SAT or SMT can be directly used to find or disprove the existence of such a set of inputs if it's a combinational design. A logical circuit network can be converted into CNF clauses in linear time [71]. If it's sequential, temporal induction can be used instead. Yosys implements this for example with the `equiv_add` command to add an equivalence declaration between two signals, `equiv_miter` to create a miter circuit, `equiv_simple` to prove equivalence of simple (combinational) designs with SAT, and `equiv_induct` to prove equivalence declarations with temporal induction. In addition, `equiv_opt` automates the usecase for verifying correcntess of optimization steps.

3.11 Conclusion

With a history of great financial losses due to unintended behavior of complex hierarchical designs, formal verification seems theoretically like a "silver bullet" to guarantee circuit compliance to specification. However, in practice, its adoption is limited. It typically has priority over functional testing and BMC only in highly complex elements like bus arbiters. This is because of its high cost: formal verification requires a specific skill set to write formal statements that truly cover the specification and lead to reasonable verification runtimes, as well as a large degree of experience with the verification tooling. For this reason, formal verification engineers are often holders of PhD titles in hardware verification—a scarce resource. For most circuits, constrained

random functional verification is deemed sufficient, even if it consumes a large amount of man power. Temporal logics are often an afterthought due to their reasonably low popularity—found bugs typically are of the sort that would have been caught with a slightly more detailed or correctly performed verification effort with simpler tools. Liveness properties are also more relevant to software, where, for example, the time to fulfill a request can vary greatly depending on its size and contents. This can be the case with coprocessors and DMA engines and the like, but moderately weaker assertions with more assumptions can discover design bugs quite well.

The support by Yosys and automation by SymbiYosys provides a flexible, adaptable toolkit for verifying safety properties. In the commercial space, a great amount of verification tooling exists with features to combat the effort required to gain sufficient faith in design correctness prior to manufacture. Famously, the typical budget for design verification has exceeded the typical design budget many years ago. Tools like Cadence JasperGold [72] and Siemens Questa [73] are capable of generating assertions for common verification tasks like bus correctness, control and status register behavior. The licensing cost of these tools varies, but estimates suggest that it's comparable to the cost of engineering time required to use them. This presents an opportunity for open source tools like SymbiYosys: bringing most of the usefulness at the same level of reliability and drastically lower prices. While SymbiYosys is licensed permissively, it is also distributed with a proprietary environment, the Tabby CAD Suite, with a SystemVerilog frontend by Verific Design Automation.

Chapter 4

Synthesis

Logic synthesis is the conversion from an RTL circuit description to a network of simple cells with a trade-off between quality metrics:

- Total number of cells, which approximately corresponds to design area and power
- Number of primitive combinational logic cells on paths between clocked cells, inversely corresponding to maximal frequency, and therefore design performance
- Synthesis runtime and memory usage

Prior to this, the source must be parsed and the syntax tree "elaborated" by converting its programming constructs like tasks, blocks, assignments, operators, and conditions into an RTL netlist implementing the language semantics. This step is also called "RTL synthesis". The semantics of if-then-else convert to signal multiplexing, non-blocking (clock-synchronous) assignments become inputs to D-flip flops, unassigned variables to D-flip flops or latches depending on the language semantics, unreachable branches and blocks are eliminated, and assignments in initial blocks become attributes of the respective signals.

4.1 Combinational synthesis

The resulting network may be limited to a set of gate types, such as AND gates and inverters, or just NAND gates, because these are all "functionally complete" sets of Boolean operators. This completeness can be proven by constructing all 16 truth tables with two inputs and one output, since from such functions, any function with any number of inputs can be built by combining them with the ITE (if-then-else) operator which also is built from 2-input operators. Constructing functions with k outputs is equivalent to building k functions with a single output.

In order to algorithmically manipulate boolean functions, they must be represented in some scheme. Truth tables are a basic representation of a

logic function, which lists the output for all combinations of inputs, and can support any number of outputs and inputs. A truth table of k inputs and l outputs requires 2^{k+l-1} bits to be stored and becomes useless as a representation for actual storage and manipulation of functions in non-trivial designs. Truth tables are an example of a canonical representation, that is, a representation that is isomorphic to the function. Therefore, truth tables are equal exactly if the functions are the same.

4.1.1 Normal forms

Another representations are conjunctive and disjunctive normal forms (CNF, also called maxterm form, and DNF, also called minterm form). A variable and a negated variable are the basic building block. A conjunction of their disjunctions then forms a CNF, and a disjunction of their conjunctions forms a DNF. The canonical DNF (canonical minterm form) can be constructed unambiguously from the truth table.

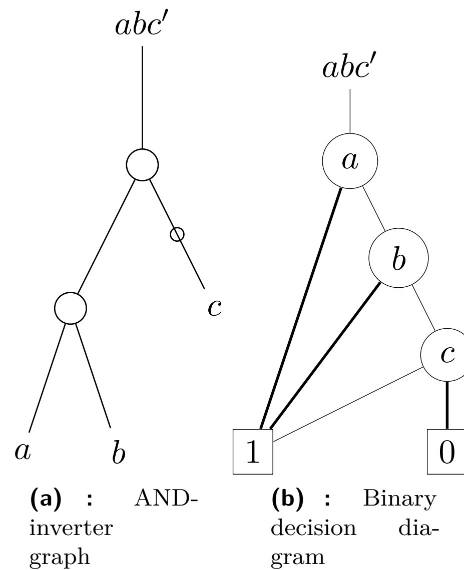
$$\begin{aligned} f\left(x_1, \ldots, x_{n-1}, x_n\right) = & f(0, \ldots, 0, 0) x_1' \ldots x_{n-1}' x_n' \\ & + f(0, \ldots, 0, 1) x_1' \ldots x_{n-1}' x_n \\ & \vdots \\ & + f(1, \ldots, 1, 1) x_1 \ldots x_{n-1} x_n . \end{aligned} \tag{4.1}$$

Similarly for the canonical CNF.

$$\begin{aligned} f\left(x_1, \ldots, x_{n-1}, x_n\right) = & \left[f(0, \ldots, 0, 0) + x_1 + \ldots + x_{n-1} + x_n\right] \\ & \cdot \left[f(0, \ldots, 0, 1) + x_1 + \cdots + x_{n-1} + x_n'\right] \\ & \vdots \\ & \cdot \left[f(1, \ldots, 1, 1) + x_1' + \cdots + x_{n-1}' + x_n'\right] . \end{aligned} \tag{4.2}$$

4.1.2 Two-level logic minimization

Any CNF or DNF formula can be expanded into a canonical version and unambiguously converted to a truth table. This shows the form's canonicity. CNF and DNF represent two levels of logic gates (three, including input inverters) and simplifying them is called two-level logic minimization and its goal is to minimize the number and length of terms in the normal form. This was more relevant in the past, when the prevalent programmable logic technology was PAL/PLA (programmable logic arrays) where inputs and their negations were brought into a programmable array of AND gates and an array of OR gates. Arbitrarily small two-level minimization is solved by the Quine-McCluskey algorithm, if such a solution exists. It has exponential time complexity in evaluating all input vectors (building a truth table) and requires solving the NP-complete unate covering problem [74] with methods like branch



(a) : AND-inverter graph

(b) : Binary decision diagram

Figure 4.1: Boolean function representing data structures

& bound with some simplification opportunities. More practical are algorithms that iteratively expand and reduce their product terms ("cubes") and remove redundant ones as they appear, such as Espresso [75]. Espresso still requires exponential operations like checking for redundancy, and contains heuristics for selecting the directions in which to expand and reduce cubes and the order in which they are iterated over. Some improvements have been made to this idea:

- BOOM [76] improves runtime on sparse functions with many inputs
- TT-Min [77] copes with dense functions with ternary trees
- EXORCISM [78] covers terms with an odd number of XOR gates, improving performance on arithmetic circuits

4.1.3 Graph-based representation and multi-level logic minimization

Binary decision diagrams [57] (BDD, see fig. 4.1b) are graphs with directed acyclic graphs (DAG) with non-leaf node out-degrees of 2. Each non-leaf node is labeled with a variable. When the variable is true, the "1-edge" is followed, and the "0-edge" otherwise. Evaluating the output of a function given a vector of inputs is done by this traversal until a terminal node is reached, which is a literal corresponding to the function's output value. Edges may be allowed to be complemented. Many BDDs can exist for a single function, but under some restrictions, the structure becomes canonical—specifically, when the ordering of the decision variables on the tree is fixed, and the function is constructed such that duplicate subgraphs aren't created. Then the structure

is called a reduced ordered BDD (ROBDD). While an ROBDD given an optimal ordering has a size linear with the function's gate count or CNF/DNF term count, finding the optimal ordering is NP-complete [79]. In general, their size is exponential. However, operations on them, otherwise difficult, like checking if a function evaluates to a constant true or false, or function equivalence, are constant in time, or linear, such as combining two BDDs into one by applying a logical operation on them. BDDs can be modified to better fit some use cases, for example, the zero-suppressed decision diagram is smaller for functions with few ones in the truth table (= long SOP terms), at the expense of adding decision nodes that each have their 1-edge and 0-edge leading to the same node.

$$f = x \cdot f_x + \bar{x} \cdot f_{\bar{x}}, \tag{4.3}$$

where

$$f_x = f|_{x \leftarrow 1}, f_{\bar{x}} = f|_{x \leftarrow 0}. \tag{4.4}$$

AND-inverter graphs (AIG, see fig. 4.1a) are also binary DAGs. Nodes are conjuctions of children, edges can be inverted, and nodes can be labeled with what function they represent. Leaves represent input variables and the constants 0 and 1. Constructing an AIG is done with structural hashing at depth 1. This guarantees that no two nodes have the same two children. However, an AIG constructed in such a way still can have redundant subgraphs, only expressed differently. To construct a functionally reduced AIG (FRAIG), [80] a conversion to a BDD or a SAT solver is used to find and merge functionally equivalent nodes and remove unnecessary subgraphs. FRAIGs are still not canonical, since one function can have multiple FRAIG implementations. AIGs don't suffer from the same size explosion problem that BDDs do. Further operations are used to simplify or reshape AIGs:

- Balancing: Nodes are merged along non-inverted edges to form N-input ANDs (covering). These are then converted back into networks of 2-input ANDs, such that the subtree depth is equal. This is equivalent to minimizing the maximum comb delay assuming equal gate propagation delays, ignoring inversion delays.
- Rewriting: Beforehand, precompute minimal AIGs for all k-input functions that only differ in input permutation, input negation, and output negation (NPN [81] equivalence class), and associate them with their hashes. The hashing function must preserve isomorphism between hash equivalence and function NPN equivalence. For a node (or each node), find a k-feasible cut. A k-feasible cut of a node is a subgraph rooted in the node with up to k inputs (subgraph leaves). Compute its function and replace the cut with the optimal cut.
- Refactoring: factoring is a algebraic method where boolean formulas are treated as polynomials

Similar data structures can be used, sometimes yielding performance and quality improvements, where ANDs are replaced, and complemented edges

can be removed, most notably majority-inverter graphs, XOR-AND-inverter graphs, and XOR-majority graphs. Their benefits vary depending on the circuit (arithmetic vs more random) and target technology (ASIC vs FPGA).

Multi-valued functions with output vectors can be represented in BDD and AIG by building a single BDD or AIG and labeling nodes representing each bit's function. Often, the behavior of a circuit is "incompletely specified", meaning the output can be true or false given a set of inputs. The representations listed need to be duplicated to model separately the "ON-set" (when is the output true) and the "DC-set" (set of inputs when the output doesn't matter). Representations that store SOP terms for linear lookup and evaluation that include "don't care"s exist, such as ternary trees, which are similar to decision trees and tries, but with don't care edges added. If only a part of a design is analyzed at a time, the preceding logic might only be able to create certain output vectors (for example, a one-hot encoder will only set one bit high) and the description of the part analyzed can be optimized with that knowledge. These are called satisfiability don't cares (SDCs) and are compatible, meaning locally computing and making use of them for optimization of one part of the logic won't affect the correctness of such optimizations in the next node. Similarly, the logic that follows might have input vectors that it isn't specified for, which makes replacing those output vectors with DCs valid. These are called observability don't cares and aren't generally compatible between "parallel" parts of the logic, but can be made compatible by doing a reverse topological order pass through the network of logic functions and replacing the functions with less specified ones [82]. While the representations so far only cover combinational logic, propagation of DCs through a design crosses clocked elements.

ABC [1] implements numerous operations, including all operations listed above, and is integrated into the Yosys [2] toolchain as the `abc` and `abc9` commands. The `abc9` command has greater timing-aware functionality. ABC uses AIGs internally for multi-level optimization to a large extent. To return to a logic network not restricted just to AND gates and inverters, the `map` command is used to create these gates in RTLIL (the Yosys internal representation language), unless mapping to FPGA LUTs or ASIC standard cells is desired (see Chapter 5.1).

4.2 Sequential synthesis

4.2.1 Finite State Machine Minimization

As stated previously, an entire synchronous circuit is a finite state machine. Designs with any amount of arithmetic have a huge reachable state space that makes it futile to do anything requiring enumerating its states. However, often even such designs include "controller" FSMs which can have only dozens of states. This presents an opportunity to make use of this knowledge to assign

state signal values to behaviorally described states such that some property is improved, for example, the resulting area, or switching power. This is especially important in always on, low power applications, like the sleep state wakeup circuitry for a microcontroller. This is called state minimization and encoding. Which signals represent an FSM state can be specified in the HDL or it can be extracted from the sequential logic. State minimization combines equivalent states into a single state. This can be done in polynomial time for a fully specified FSM. Hopcroft's algorithm [83] performs it in $O(n \log n)$ by merging each Nerode congruence equivalence class [84] [1] into a single state. The solution is unique as no arbitrary choices are required. If an FSM has unspecified behavior given some input symbol in some state, it's incompletely specified. This is a situation where the designer has (hopefully deliberately) specified that in that state such an input symbol is guaranteed not to arrive. The behavior of the FSM is then allowed to be chosen by the synthesis tool. State minimization on incompletely specified FSMs is NP-hard. Two states are said to be "compatible" if both of the following holds for each possible input:

- Their output DCs can be set to concrete values such that their outputs are all equal
- All their successor states are compatible

If two states are compatible, we can merge them without violating the FSM specification. Since there may be many such operations, we want to find the "maximum compatibility sets" by selecting pairs to merge such that the number of states is minimal. This can't be done greedily, because compatibility isn't an equivalence relation. It isn't transitive: we can't generally replace state A with state B and then state C, even if A is compatible with both B and C. As an example, let state A have unspecified transitions, B some specified transitions, and C different from both A and B. While this popular definition is symmetric, we can't replace a more specified state with a less specified one. Plotting the states and their compatibility relations as a graph, we find ourselves wanting to find a clique and replace it with a single node, then another, such that the final number of nodes is minimal. The choice of which clique to replace makes this algorithm NP-hard. STAMINA [85] is an algorithm which solves this problem exactly as well as through an inexact heuristic mode.

4.2.2 Retiming

Retiming is a technique to reduce the critical paths or area by moving registers around. For example, if a 2-input logic gate has each of its input driven by a register, these registers can be removed, and as long as a register is added on the output signal, the circuit function is unchanged. The gate has been

[1] This is the same equivalence as in the Myhill-Nerode theorem used to determine if a language is regular.

removed from the timing path containing the downstream gate, and added to the paths containing the upstream gates. To formalize this process, we construct a weighted graph, where nodes represent gates and edges represent connections. Nodes are weighed with approximate propagation times, and edges with the number of registers on the connection. A retiming operation is defined as a function assigning an integer to each node. This (often negative) integer determines the number of registers to move from inputs to outputs. Performing this retiming changes the weights of each edge (u, v) by adding $r(u) - r(v)$. The retiming is valid when none of the resulting edge weights are negative. The goal of minimizing critical path length can now be formulated as minimizing the path with zero weight edges and the largest sum node weights. This seems like a branch-and-bound problem, but turns out to have polynomial solutions [86].

ABC [1] supports the `retime` operation, which implements linear-time retiming given a target period [87], as well as a heuristic critical path reducing algorithm, as well as an area minimizing algorithm. ABC finds the initial register states with SAT, since retiming affects these.

4.3 Physical synthesis

In an ASIC PDK, often several implementations of the same gate are available, differing in size and drive strength. At the expense of power and intrinsic delay, high fanout[2] nets can be driven with larger gates to decrease the propagation delay. Using larger gates can also increase the delay due to larger intrinsic capacitance of the output, and negatively affects the input net timing because its input is harder to drive as well. Alternatively, the output of a gate can be buffered, which doesn't have this drawback, but typically increases the area and power more than selecting a larger gate. Gates can also be duplicated, splitting the fanout.

[2] Fanout is the count of signal sinks connected to the source. High fanout nets are typically control signals, like clocks and resets.

Chapter 5

Physical design automation

After a digital circuit meant for integrated circuit fabrication is designed and verified, a good amount of processing is required to reliably bring it from RTL netlist to physical geometry of each fabrication layer. Similar to PCB GERBER files, this data is captured in GDS or Aperture file formats. A well-integrated example of such a full "design flow" is OpenLane [88], which automates together several open source tools and programs together with the place-and-route flow by OpenROAD [5].

Each of the presented components presents a set of algorithmic challenges, some of which are quite unique. Throughout this chapter, circuits designed with the SkyWater 130 nm open source process development kit (PDK) will be used for illustration.

5.1 Synthesis backends

A common approach to physical ASIC design is the standard cell design style. The designer is provided with a "cell library", defining a set of cells for combinational logic, flip-flops, as well as cells that aren't tied to the logic design, such as antenna diodes[1], tap cells for mitigating CMOS latch-up, decap cells adding capacitance to the power rails to mitigate voltage drops and ground bounce, delay cells to meet timing. Standard cells are optimized for the manufacturing process as well as the target "corner" in the PPA trade-off. Standard cell design assumes a fixed width for the cells to allow for a regular routing structure, constraining and simplifying placement and routing. It increases the need for routing over fully custom designs, but this problem has been reduced with advancements in fabrication processes allowing for more metal interconnect layers.

A similar problem in physical design automation for FPGAs is the mapping of logic networks into its fundamental combinational logic building blocks, the look-up-table (LUT), which is a reprogrammable read-only memory with a k-bit data bus, indexed by a k-bit address. The memory is programmed with

[1] Antenna diodes mitigate charge buildup on long connections in the fabrication process.

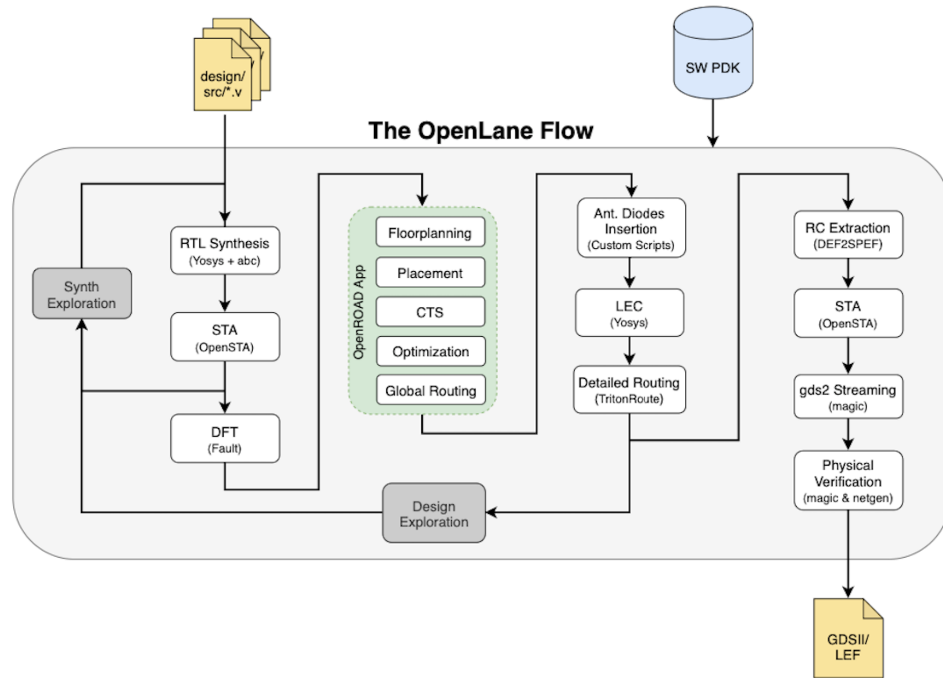


Figure 5.1: OpenLane design flow [88]

the truth table of a logic function. The address bus becomes the function's input vector. The inputs and outputs of these basic elements of logic (BEL, typically also contain registers or adders and multipliers) are then routed to neighboring cells through multiplexers.

A tech mapping Verilog file for a cell is marked to be compatible with a cell in RTLIL, and when elaborated, generates a structure made from simpler tech specific primitives. This mapping can be conditional, for example, if the tech library only specifies cells for some bit widths and not others. As an example, the file `pdks/sky130A/libs.tech/openlane/sky130_fd_sc_hvl/rca_map.v` in (correctly setup) OpenLane is marked to match `$add` Yosys blocks to single bit full adder technology specific primitives.

Multiple RTLIL cells aren't merged at this stage. If mapping to larger primitives is necessary, it needs to be explicitly instantiated in the original design, or the `extract` run can be used, which searches the design for subcircuits from a given list, creating a custom cell types. The `extract` can also be used without a subcircuit list to "mine" for common subcircuits, which can be used as information for what cells to additionally add to the cell library to meet PPA targets. Tech mapping isn't responsible for generating SRAM cells and encountered memories will be converted to flip flops, which is much less efficient. Often, not all D flip flop (DFF) cells are implemented in the cell library. The `dfflibmap` pass converts unimplemented DFF cells from implemented ones by invoking `dfflegalize`, which transforms DFFs by adding inverters or hard-wiring more complex DFFs to act as simpler ones. Afterwards, `abc` is

used for logic optimization and AIG-based to map combinational logic cells, unlike in FPGA techmapping, where the design is mapped with the `techmap` pass, after `abc` optimizes the design and performs generic LUT mapping, only given the LUT width. For FPGA designs, DSP blocks are converted to target-specific nodes with `techmap` before the generic `techmap` passes, similarly to for example full adder cells in a PDK. The FlowMap [89] algorithm is a classical LUT mapping algorithm, which finds "k-feasible cuts" for all AIG nodes and selects cuts that minimize delay. The `if` LUT mapper in abc brings improvements, most notably, linear runtime with AIG size rather than polynomial, and only selecting "priority cuts", allowing for a trade-off between delay and area. Using `abc` to map to standard cells internally works the same way. This assumes cells have load-independent (regardless of fanout and parasitics) timing and relies on gate sizing and fanout optimization (see Chapter 4.3) to meet this assumption.

The liberty file format defines propagation delays, fanout limits, resistive, capacitive, and leakage properties, discretized rise/fall waveforms, as well as power characteristics and operating conditions, all with relation to its input, output, and clock pins.

The minimal example provided by Yosys documentation [2] is shown in Listing 3. OpenLane extends this with extra `techmap` calls.

```
# convert design to (logical) gate-level netlists
techmap
# perform some simple optimizations
opt
# map internal register types to the ones from the cell library
dfflibmap -liberty cells.lib
# use ABC to map remaining logic to cells from the cell library
abc -liberty cells.lib
```

Listing 3: Example Yosys script

5.2 Floorplanning

The floorplanning phase places cells from a netlist into 2D space with several considerations determining placement quality. If all I/O pads and macro cells and macro cells (blackboxed, pre-routed submodules of a known size) are fixed in place, we consider the task at hand "placement". If some of them have flexible dimensions or locations, we consider the additional task prior to or integrated with placement "floorplanning".

Before floorplanning is done, it's often beneficial to introduce artificial hierarchy to the "flat" netlist we take as input. This is called partitioning. Partitioning is the decomposition of cells into subsets such that connections crossing subset boundaries are minimized. This is without any notion of 2D

space yet. The desired size of the subsets needs to be set large enough to allow for placement and routing quality, but small enough to reduce runtime and memory usage, since these factors are important for pushing the limit of maximum feasible circuit complexity given the desired design iteration duration—often the ideal target is for silicon rebuilds to be done nightly, meaning under 24 hours. Additional constraints for partitioning include minimizing the maximum number of times a path is cut (called the delay), and limits on the terminal count per partition, and on the partition count.

Iterative top-down partitioning works by cutting partitions (sets of nodes) into equally sized smaller partitions by selecting k lines through the 2D space such that the number of edges crossing the cut is minimized. The Kernighan and Lin (KL) [90] algorithm modifies this by generating a random balanced partitioning first, and swapping pairs of nodes between partitions, with pairs selected by inspecting the number of cut crossing edges removed and added and picking the "best" pair. Swapped pairs are then locked in place and not swapped again. Once there are no more improvements to be made, all nodes are unlocked, and another iteration is done. These partitioners only minimize the number of edges cut. The KL algorithm has cubic complexity and is guaranteed to find partitions balanced in area as a trivial invariant of cell swaps. The Fiduccia-Mattheyses (FM) [91] algorithm, a modified versions of KL, uses cell moves instead of swaps, operates on hypergraphs with a specific representation, and achieves quadratic complexity, while honoring a set partition area balance constraint. The hypergraph is a graph-like mathematical structure, practical for the problem of design partitioning. A hypergraph has a set of nodes similar to an ordinary graph, but its edges are sets of any positive number of nodes, rather than only 2. A netlist can be represented as a hypergraph with components (cells) as nodes and nets as hyperedges. Whenever a hypergraph needs to be treated as an ordinary graph, its hyperedges need to be replaced with cliques or star graphs.

There are many more approaches to top-down partitioning. One inspects the spectrum (set of eigenvectors) of the Laplacian matrix (difference of the degree matrix and the adjacency matrix of the graph) and uses eigenvectors as 1-dimensional node placement that then can be partitioned easily.

Conversely to top-down partitioning, bottom-up clustering (see fig. 5.2) works by joining partitions to find their shared parent partition. These partitioners typically aim to minimize delay added by the partitioning. Delay optimality is achieved, if the maximum estimated delay summed over a path from a primary input (PI) to a primary output (PO) is minimal. In the general delay model, the node delay, intra-cluster path delay, and inter-cluster path delay estimates are required.

The Rajamaran and Wong clustering algorithm [93] operates on a circuit DAG and achieves delay optimality under the general delay model. First is the labeling phase, where nodes are labeled in topological order with the maximum delay from PI to the node, using the general delay model. As this is done for each node, a suggested cluster of predecessors of the given node is

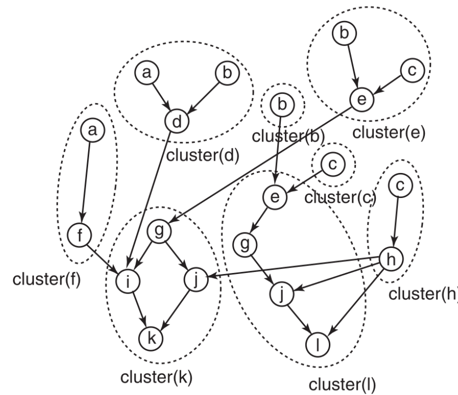


Figure 5.2: Clustered graph adapted [92]

constructed as the set of predecessors furthest away from the PI, therefore closest to the given node. Afterwards, POs are added to a set L. As long as there are nodes in L, we remove a node from L, the suggested cluster of the node becomes a cluster if it isn't a cluster already, and all input nodes for the cluster are added to L.

The LUT-mapping tool FlowMap [89] from the FPGA world may be used for clustering as well. The difference in results is that FlowMap restricts the maximum number of inputs and outputs to a cluster to a set number, since it's meant to merge LUTs to fit into k-input LUTs. This is called the pin constraint. Also, FlowMap uses the unit delay model, which differs in that it only considers gate delays. An implementation of FlowMap is available in Yosys as the `flowmap` pass.

Multi-level coarsening then chains several clustering and partitioning runs to create a hierarchical partitioning with improvements in runtime. The hMetis algorithm [94] first runs k levels of clustering. This creates a small network of large clusters. This network is then partitioned without cutting the clusters. The clusters are unclustered within each partition, revealing more nodes. Each partition is now again partitioned, and so on, until the lowest level clusters are unclustered and partitioned. This process is called decomposition and refinement. We arrive at the final result, a hierarchically well-partitioned circuit. The clustering and decomposition allow the program to only hold in memory as much circuit complexity as needed for good partitioning results, reducing runtime and memory requirements. The coarsening steps also don't have to be optimal with regards to delay, as they only server to localize interconnected circuit subsets for better partitioning. Therefore, the coarsening is done in a faster way than typical clustering.

Additional features are added to some of these algorithms to match industrial use-cases, such as register retiming, power dissipation considerations, and semi-manual partitioning according to timing criticality and clock domains.

5.3 Placing

A placer assigns cells positions on the die, such that total wire length and area are minimized. While actual wire length is going to be decided by the router after placement, estimation models are used instead. The minimum Steiner tree (similar to minimum spanning tree of a graph with edges weighted with cell distances, but with adding extra vertices allowed) gives a lower bound for a single net, so the sum can be used as the objective function. The optimization version of the Steiner tree problem is NP-hard, so instead, half-perimeter wire length (HPWL) or minimum spanning tree can be used instead. HPWL is given by the perimeter of the smallest bounding box covering all pins in the net and be computed in linear time with respect to pin count and empirically correlates with minimum Steiner tree length. In practice, other estimations are often used to drive placer decision-making.

If meeting timing is implemented as a constraint, connections are assigned timing budgets that according to the estimated timing budget of a path. Timing optimization throughout the design process isn't only meant to eliminate negative slacks, but also to minimize positive slacks, allowing for solutions with smaller area and power. If positive slack is available on some connections on a path, while another connection has negative slack, it is possible to redistribute the timing budget to meet timing without necessarily immediately modifying the solution. An alternative approach is reweighting the edges according to their criticality and basically optimizing slack as a part of the placer objective function. This leaves more opportunity for intermediate placer states that violate timing, but converge to meeting it, and leads to better runtimes and memory usage as the design grows because the number of paths to check for violated timing can grow exponentially with the number of gates.

Congestion is a factor not covered by the wire length estimates. When each of many nets would ideally be routed through the same area, that area is said to be congested. If large amounts of congestion are present in the placed design, then the router is forced to route connections through long bypasses, adding delay to critical paths, and its runtime can be much longer because of the difficulty of finding satisfactory solutions.

Min-cut placement [95] (see fig. 5.3) is a method of placing via partitioning in 2D space. This requires cutting geometrically localized partitions into smaller partitions along a horizontal or vertical line, until the partitions are of a desired size, which can be as small as one cell. If it's larger than one cell, a process called legalization is required to finish placement, aligning cells to a desired structure (snapping to rows, see fig. 5.4) without overlaps. This is often implemented as a part of a "detailed placer" which does additional local optimizations. The main stage is then called a "global placer".

The cut direction (horizontal or vertical) can be selected in an alternating order or depending on the partition aspect ratio. In this approach, only

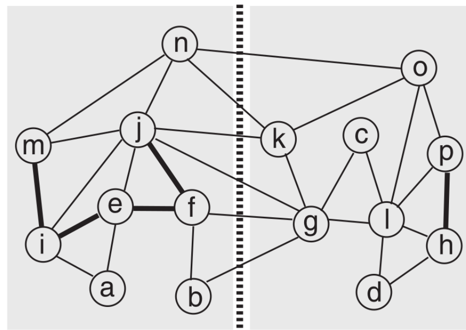


Figure 5.3: Min-cut placement, adapted [92]

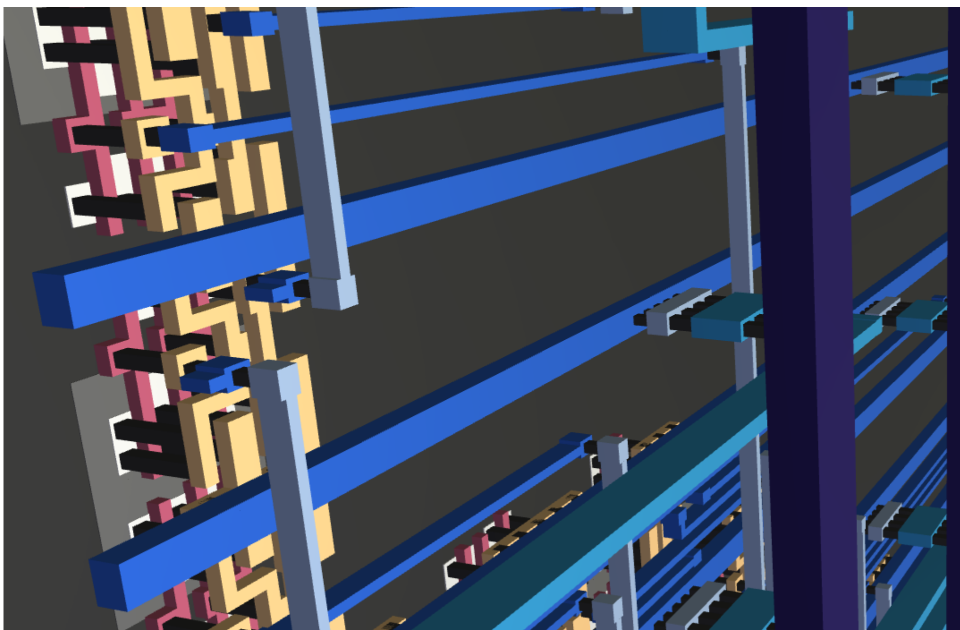


Figure 5.4: VCC and GND rails in a standard cell row

hyperedges between nodes within a partition are considered. “Global” edge lengths can be reduced by considering nodes in partitions other than the one being cut, and swapping them to one of the closest partitions. Alternatively, a set of adjacent partitions can be considered at a time, resulting in a “sliding window” approach, which can improve performance. If congestion is present at the end of placing, free space left in the partitions can be redistributed by swapping cells further, ironing out congested areas.

Simulated annealing is a general technique popular for finding near-optimal solutions to combinational problems like the traveling salesman problem. An initial random input is generated, and a temporary “temperature” variable is initialized. Iteratively, random alterations are considered. When the objective function shows an improvement, the alteration is accepted, otherwise, it’s randomly accepted with a probability depending on the temperature and objective function change. The temperature is decreased by an amount

generally non-linearly depending on the objective function value, and another iteration is performed. Accepting worse solutions allows the method to escape local minima and solve very non-linear problems. When the temperature reaches some threshold, the loop is terminated, and small improvements can be iteratively reached while available. Simulated annealing is applied to placing by encoding actions like cell swaps, moves, and rotations. A notable example of using simulated annealing for placement is TimberWolf [96], which has since its conception been enhanced with multi-level clustering as a part of the simulated annealing procedure.

Another class, which has gained popularity with great performance, are analytical placers. Initially, absolute value wire length models were approximated with quadratic ones to turn the objective function into a continually differentiable convex function, which can be easily minimized by analytically finding its gradient ahead of time. It should be noted that these objective functions separate the x and y components of the wire length model. Minimizing their sum is then done using linear methods such as conjugate gradient descent or Nesterov accelerated gradient descent. The quadratic approximation doesn't lead to minimal wire lengths, so non-linear approximations were introduced, empirically converging to better solutions. Analytical placers require the hypergraph to be flattened into a graph, converting each node (net) typically to either a star or clique graph of size equivalent to the net's pin count. The wire length is computed from this flattened graph. Good performance has been achieved with the Bound2bound net model [97], where, for example for the x-component, all pins are connected to the leftmost and rightmost pins in the net. The connections are then weighted with

$$\frac{(x_i - x_j)^2}{(p-1)\,|x_i - x_j|}, \tag{5.1}$$

where p is the net pin count, x_i and x_j are the coordinates of the pins. The denominator linearizes the wire length, leading to direct HPWL optimization, while retaining desirable gradient properties. One major difference is that while partitioning and simulated annealing guarantee gates don't overlap since their locations are constrained and they are typically swapped between valid locations, analytical placers don't come with such a guarantee, and often require some post-processing to push them away from each other, or additional expressions in the objective function to minimize overlap in each iteration.

A peculiar type of analytical placers are force-directed electrostatic placers [98], [99], which measure placement congestion (density) as the potential energy of a system of charged particles. Each node is given a charge equivalent to its area. The density gradient then corresponds to electrostatic repulsive force. The resulting problem is solving Poisson's equations with Neumann boundary conditions. To solve it, we need the potential function to be smooth, which can be achieved by performing smoothing with a bell-shaped function of some kind, or by decomposing to a limited set of discrete spectral components. Smooth spreading means cells aren't swapped to cope with

congestion, but only pushed away from each other. Penalizing density in the objective function is equivalent to setting max density as a constraint by introducing a Lagrangian multiplier, the density penalty factor.

The global placer in OpenROAD [5], a rewrite of the state-of-the-art RePlAce [100], implements an electrostatics-based analytical placer using the spectral method, and includes a GPU solver implemented in CUDA. The GPU solver is unused in the OpenLane flow due to OpenLane's distribution as a Docker container. RePlAce also includes several optimizations that make it competitive in standard benchmarks and large real world designs:

- Solver step size is automatically adjusted while the solver runs
- When step size is overestimated, the solver backtracks to an earlier state and adjusts
- The density estimation used is obtained by calling the global router for one iteration, getting its congestion in each "bin" (rectangle area in a rough grid) and inflating all cells corresponding to the density and density penalty factor, almost guaranteeing routability
- The density penalty factor is increased as the placer improves wire length over its iterations. This allows connections between cells to play a greater role at the start of the placement without cells in a strongly connected part of the design getting "stuck" on the repulsive force of other cells in the way
- Whenever the sum overflow (excess cell area in a bin) over all bins improves by a set amount, routing slacks are estimated, and an amount of the worst offending nets are reweighted. This gives the wire length of critical paths priority in the wire length minimization

The global router routing congestion estimation step dominates the global placer runtime on larger designs.

The detailed placer in OpenROAD is OpenDP [101]. OpenDP legalizes cell placement by searching for the nearest (in terms of Manhattan distance) legal place to move a cell to. OpenDP can also run simulated annealing to minimize the total displacement needed to find legal placement, which isn't implemented in OpenROAD's implementation. It supports cells that have mixed height (spanning more than two rows).

5.4 Clock tree synthesis

Only once a final placement is known for all logic cells, the clock tree has to be created due to the connection length dependence of the required clock signal buffering and delay matching. Also, since clock signals are a great source of obstructions in the routing phase, clock tree synthesis (CTS) can't be performed after routing is done, because good routing takes into account all (at least locally) required connections. The skew-minimal clock tree is

an H-tree, an H-shaped line fractal, which reaches every point in a square area with a path from its center with equal distance. Equal delay to clocked cells is important for timing closure, because clock skew reduces the slack for combinational logic propagation delay between clocked cells, reducing design performance, increasing down router runtime, or outright causing the router to fail to find a valid solution. Furthermore, it's important for chip yield due to variation in propagation delay due to physical inconsistencies in the manufacturing process. Also, clock skew can cause hold violations, which can't be fixed by reducing the clock frequency[2].

However, needlessly providing a clock to all points in a sparsely utilized region increases power consumption with buffer switching activity and complicates routing of other signals. In real-world designs, clock trees consume a significant portion of total design power. A topology that decreases clock tree wire length at the cost of greater skew is the fishbone topology, with a single trunk in one direction, and a set of perpendicular wires to cells. The permissible slack is also not a global parameter, but varies among combinational paths. A trade off between H-tree and fishbone clock trees can be made by using a generalized H-tree [102], which is a balanced H-tree with arbitrary branching depths (number of branching points per path to leaf) and numbers of branches at branching points.

The CTS tool in OpenROAD is TritonCTS. It hierarchically uses iterative k-means clustering to find branching points for a set of sinks—this set being one of the clusters found earlier. Buffers are then inserted as needed, not necessarily at every branching point of the clock tree.

5.5 Routing

The basic formulation of the routing problem is that for finding a path between two points in a grid with obstructions. The simplest optimal solution (finding the shortest paths) is Lee's algorithm [103], which iteratively in step i marks all grid points reachable from the source in i steps with the least number of steps (an integer less than or equal to i) and a list of grid points the point is reachable from in that number of steps. This propagates a "wave" through the grid, until the sink is reached. The shortest path which reaches the sink is now retraced by traversing the lists from the sink to the source. Its asymptotic complexity is $O(hw)$, which it reaches with long paths, which force the wave to propagate throughout the entire grid. Soukup's algorithm [104] improves this by considering only steps that bring it closer to its target if possible. While it has empirically much better runtimes, it has the same time complexity, and isn't guaranteed to find optimal routes (it might greedily go through a maze rather than walking around it). The well-known A* algorithm

[2]This happened to the entire first batch of Google's OpenMPW program, MPW1, due to the CTS failing to provide a delay-balanced tree, and the lack of parasitics extraction for routed wires. Timing was estimated before routing, from placement. This has now been addressed by the OpenROAD parasitics extraction tool OpenRCX.

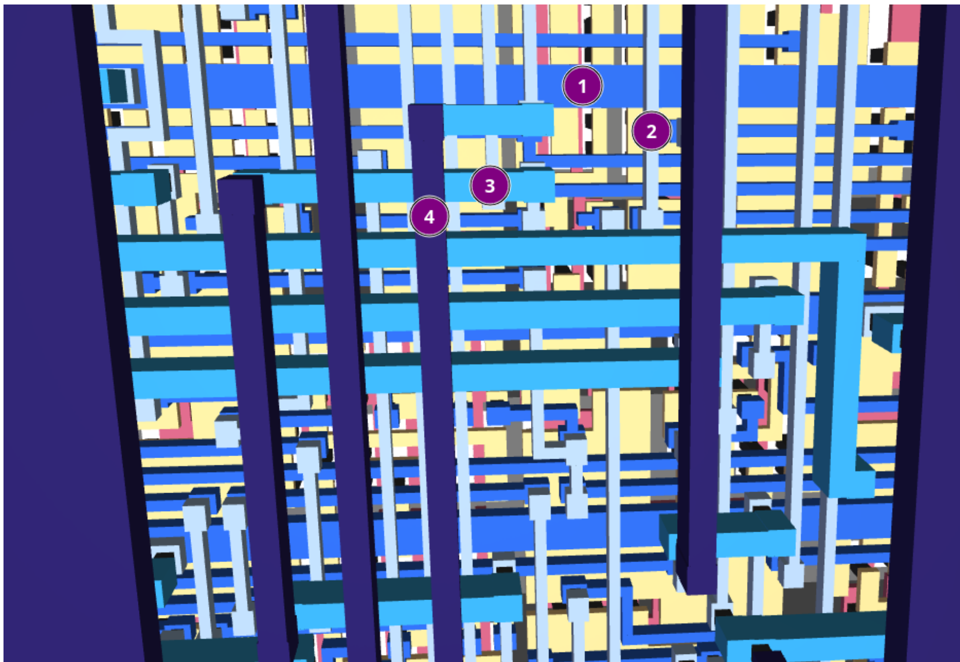


Figure 5.5: Four routing layers with alternating direction

[105] is a modification of Dijkstra's algorithm for finding the best path to a single goal, using time-optimal heuristics. Hadlock's algorithm [106] uses A* with heuristics based on the "detour number", the number of steps taken that don't decrease the distance from the destination. It's guaranteed to find the optimal solution, also has the same time complexity, but empirically has better run times than either previous solutions.

All solutions mentioned so far are "maze routing" algorithms, which start at one point, and search for the other, traversing the grid. Notable improvements can be made by searching from both ends cooperatively, taking leaps spanning many grid steps at a time. This is how "line-probe" algorithms operate. In the initial step, a pair of line segments (vertical and horizontal) are emitted from the source as well as the terminal. These terminate either at the grid boundary, or at grid points immediately before obstacles in the way, and are labeled with depth 0. Iteratively, additional line segments perpendicular to existing ones are added, with increasing depths, while marking whether they originated from the source or terminal. Once a source segment crosses a terminal segment, a path can be constructed from these. This general scheme has been originally implemented in two ways. The Mikami-Tabuchi algorithm [107] creates new perpendicular segments at all points of existing lines, while the Hightower algorithm [108] only creates them at points immediately before obstacles, or immediately after an obstacle is no longer running parallel to a segment. Both algorithms have the same time complexity, which is linear with the number of line segments in their found solutions. Mikami-Tabuchi requires more work to find a path, but is guaranteed to find the optimal one, while Hightower has no guarantees of finding an optimal solution or even a

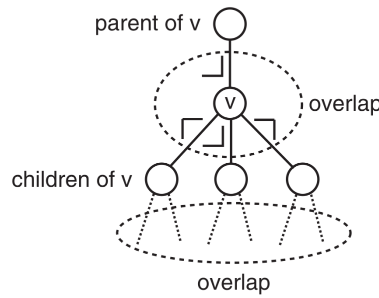


Figure 5.6: L-RST separability, adapted [92]

solution at all.

As previously noted, the minimum Steiner tree gives a wire length lower bound for a single net. Multi-terminal routing algorithms attempt to find the routing solution for a given net with minimal wire length or delay. These algorithms do not consider obstructions, but are still very useful, as will be explained later. To simplify the problem, we will consider rectilinear Steiner trees (RST), meaning restricted to horizontal and vertical wire directions. Since the minimum spanning tree (MST) can be computed in linear time, given a complete graph weighed by terminal (cell pin) distances. Such a graph is a clique and can be built in quadratic time. Rectilinear MST (R-MST) is sufficient to find a routing with at most $\frac{3}{2}$ the wire length of an optimal RST [109]. To optimally route a net, it's unnecessary to consider all possible grid points. This is because all optimal solutions have edges on a strict subset of the grid, the Hanan grid [110], which includes only grid rows and columns with at least one node present.

The L-RST algorithm [111] (see fig. 5.6) finds an MST, and assigns whether an "upper L" or "lower L" shape is used to connect the endpoints of an edge in the MST. It achieves this in linear time by building an R-MST from a separable MST, meaning only adjacent edges can overlap in the routing grid. This is guaranteed by making use of the freedom whenever a conventional MST algorithm has multiple choices. This is implemented by weighing edges with a 3-tuple rather than just Manhattan distances, and sorting lexicographically to make optimal choices. When distances are equal, vertically taller edges are preferred. When their y-distances are equal as well, the edge with the rightmost node is selected. Once the separable MST is found, a root is arbitrarily selected, creating a proper tree to be traversed bottom-up. In this traversal, we calculate the length of overlaps given the edge to parent is routed as upper L as well as the length given it's routed as lower L. As each non-leaf node is considered, overlaps are counted for each of the 2^d choices for routing only the edges incident to the node. To this, we add previously found overlaps in the subtrees rooted in the considered node, given the routing selected for the edge to each child node. Since d is the degree of the MST node, which is bounded in the context of distance graphs in a 2D plane [112], we are doing constant work on each node, so this bottom-up pass is linear.

All this is followed by a linear top-down pass, which finally assigns routing to edges in such a way that overlaps are maximized, finding the minimal R-MST.

Alternatively, we can aim to directly build a good RST. The simplest way is by adding Steiner points one-by-one, which is called the 1-Steiner problem [113] in considering all possible Steiner points to add and computing the wire length improvement (“gain”). The implementation is as follows: starting with an MST, for each MST node and edge pair (such that the node isn’t an endpoint of the edge) consider adding a Steiner point to the closest point to the node on the rectilinear bounding box of the edge. This adds a cycle to the graph. Disconnect the cycle by removing its most expensive edge. The wire length gain is the length of the added edge subtracted from the length of the removed edge. All such pairs are considered, sorted by gain, and their suggested Steiner points accepted, if previous Steiner points haven’t made them infeasible. Whenever a node and edge pair is selected as the best options, the edge is snapped to the upper or lower L routing defined by whichever point is closest. Therefore, only as many Steiner points as there are edges can be added. Overall, this modification [114] of the 1-Steiner algorithm is quadratic. However, this doesn’t generally generate the minimum RST, since locally the best Steiner point to select isn’t necessarily present in the globally best set of Steiner points to add. Solving the RST problem is in fact NP-hard, but 1-Steiner does build shorter trees than L-RST, at the price of a longer run time.

The above algorithms minimize total added wire length. However, it’s possible that their solutions contain a long route between two points that exceeds slack in an associated critical path. Instead, the longest output-input path in the net, called the radius, can be minimized to avoid this situation. Achieving this requires leaving the optimal Steiner trees and formulating the bounded radius minimum routing tree (BR-MRT) problem, where the minimum wire length routing is constructed given a radius constraint. An easily implemented solution is the bounded Prim algorithm (BPRIM). The MST search is started from the single output (source) present in the net. In general, there may be tri-state driven pins in a circuit, but are a special case that can be expensively handled. Random logic is never modeled as tri-state. Whenever an edge is considered for addition, a relaxed version of the bounded radius invariant is checked. The relaxation constant is specified by the user and sets the trade off between meeting the radius bound and minimizing wire length. If the condition isn’t met, the point is instead connected to a point in the MST that meets the unrelaxed condition. The edge used to connect it is called an “appropriate” edge. The radius and wire length both have known bounds given the given relaxation parameter. The minimal radius is the Manhattan distance from the source to the furthest sink, which can be computed and achieved by setting the relaxation parameter to guarantee the radius bound constraint. Solutions found by BPRIM often have needlessly long appropriate edges with crossings. An alternative method is the bounded radius bounded cost (BRBC) algorithm. First, the MST is rooted in the source and labels are

stored in a list in order of depth-first traversal, with non-leaf nodes present multiple times. The solution is initialized to the MST. We traverse this list. The weight of the edge traversed to get to the next node is added to the integer, unless it's greater than the distance from the source multiplied by the relaxation constant, in which case we reset the integer to zero and to the solution we add an edge from the source to the node. We now have a graph with cycles. Its shortest path tree is our BR-MRT, which we can obtain in cubic time.

Whether wire length or radius should be prioritized can be estimated with Elmore delay [115]. From resistive and capacitive properties of connections and nodes, an estimate well correlated with the signal propagation delay can be constructed. Routers can directly optimize for the Elmore delay. The delay to propagate a signal through a tree where edges represent routing and nodes represent their forks is formulated as a function of the total capacitance driven in each subtree, the resistance and capacitance of each edge on the path through the graph. In the case of a single long connection, the Elmore delay becomes quadratic with respect to its length. In Elmore routing (ERT) [116], we root a tree in the source, and iteratively add nodes, such that the maximum (over all leaves) Elmore delay is minimal. Nodes to consider are limited to the nearest neighbors of nodes already in the tree. Notice that we don't add any Steiner points, only finding the Elmore-optimal MST. This is changed in Steiner Elmore routing (SERT) [117], where we search for nearest nodes to edges already in the tree rather than to tree nodes, adding Steiner points when to build the connection.

While considering each net in isolation is possible, it doesn't yield the best results possible. When an infeasible routing is found, which is typically the case, the offending nets need to be "ripped up and rerouted" (R&R) to alleviate congestion. Solving multiple nets concurrently can help reduce congestion directly. One option is formulating routing multiple nets as a multicommodity flow problem. Conversions to integer linear programming (ILP) are a common tool in finding solutions to various formulations. However, prohibitive runtime has been a major drawback. In negotiation-based routers [118], each node available as a routing resource is associated with a cost based on its congestion history in previous iterations and number of signals using it. As the congestion history term increases for congested nodes, nets need to negotiate over access with priority depending on their slack. This is a form of R&R routing.

In reality, routing isn't a 2D problem. With several metal layers available, the goal becomes finding a valid short timing-based routing solution through the 3D space, such that the via count is also minimal. Via minimization is important for yields and failure rates. Directly searching for routes through the 3D space is computationally infeasible, so 2D global routing solutions are assigned to layers. However, building Steiner trees and assigning paths to layers can be implemented with methods that decrease the number of vias required.

Similarly to placement, routing is typically split into a global and detailed phase. Global routers assign each terminal of a placed cell to a tile in a grid, and search for routes through this small new grid. Many nets can pass through or have terminals in a single tile, so each net can be routed almost separately with Steiner tree methods, despite the fact that they do not consider any obstructions. Cross point assignment (CPA) determines where on the tile boundaries the nets will pass through. Detailed routing then connects nets together within each tile, connecting terminals and cross points. Multi-net methods need to be used at this stage, working on each tile separately. The detailed router must be able to route the design correctly with respect to all rules that guarantee manufacturability, which are later verified with design rule checking (DRC) tool.

The global router in OpenROAD is FastRoute [119]. First, minimal RSTs are built. To minimize vias and move routing segments away from congested areas, opportunities for arbitrary segment moves ("segment shifting") are used. The Hanan grid isn't used as a restriction for this. Even before this, congestion isn't estimated from the solutions directly, but for each L shape, both options (upper L or lower L) are applied with half weight. In real world designs, routing demand for horizontal is often higher than vertical or vice versa, and this can also be used as a heuristic for selecting minimal RST implementations, to trade off vertical routing for horizontal. Also, the maximum routing demand for a row or column is minimized, leading to the preference of for example two shorter vertical segments at different horizontal coordinates than a single longer one.

The detailed router in OpenROAD is TritonRoute 4.1 [120]. A detailed router needs to be aware of DRC rules to find valid routes. To implement this at speed, look-up tables between vias and track (connections with fixed width, unlike abstract connections) L-turns. The vias used in detailed routing are selected according to heuristics that aim to reduce the amount of influence (obstruction) the via has on its surroundings, given the track orientations. This is done separately for regular routing and "pin access". Standard cells have defined pin shapes on the first metal layer (often called local interconnect, li1 in SKY130). This layer can't be used for routing. Pin access [121] is the problem of finding a valid point on each pin to add a via to the routing layers and is done as a preprocessing step before detailed routing, which then sees those vias as the routing targets. TritonRoute finds access points while ensuring there is a valid direction the track on the first routing layer will then likely be routed to, including the "up" direction, which corresponds to an immediate via to the second routing layer. Ignoring this consideration would lead to pin access creating unroutable points. The detailed routing procedure itself is then performed as sequential A* R&R, which is penalized, when a DRC violation look-up table matches the situation.

5.6 Static timing analysis

The propagation delay between FF in a functional circuit has both an upper and lower bound defined by the hold and setup times, desired frequency, and clock skew. Formally, with $t_{cycle} = \frac{1}{f}$:

$$t_{cycle} \geq t_{delay} + t_{setup} + t_{skew}, t_{delay} \geq t_{hold} + t_{skew}. \tag{5.2}$$

This must hold for all paths that are possibly active. For example, if states on a path was guaranteed not to affect the input signal of any FF because, for example, that would require two outputs of a one-hot encoder to be true at once, it could be ignored in timing analysis. This would make it a "false path". However, only "static" timing analysis (STA) is typically done, which doesn't try to determine whether a path is "sensitized" due to the complexity that would add.

To meet timing constraints (achieve "timing closure"), estimates as accurate as possible must inform the design process, from synthesis to routing, using the newly created specific information. For example, clock skew is unknown until CTS, and can't be left out after CTS. The combinational logic in a circuit for the purpose of CTS is represented as a DAG with all FF outputs as sources and inputs as sinks. Nodes (gates) as well as edges (routing connections) are weighed with their propagation delay estimates. Checking for setup violations requires finding the longest path through the DAG, which can be done with a simple traversal in topological order, assigning signal arrival times to the nodes on the way as it's traversed.

Rising and falling signal propagation often differs by significant enough amounts to warrant separate characterization and consideration in arrival time and slack calculation. Propagation times aren't fully deterministic, can be influenced by surrounding switching activity, and can be modeled with probability distributions. The basic approach is to use worst-case characteristics for each step, but for optimizing PPA metrics, it's possible to directly model the slack probability distribution, trying to meet timing for a set low violation probability, and to take into account surrounding signals.

Hold violations are guaranteed not to happen by CTS by buffering and balancing the tree. However, hold conditions must be rechecked when the circuit is modified in physical synthesis.

5.7 Layout vs schematic checking

Since bugs may be present in the physical design flow, the standard cell netlist may implement a behavior different from that specified in the original RTL description of the circuit. Layout vs schematic checking (LVS) is a step that ensures consistency or helps find the step which creates inconsistency. In

OpenLane, the netgen LVS tool from the QFlow [122] project is supported to perform this taks, as well as the LVS functionality of the KLayout layout viewer [123].

5.8 Circuit extraction

A placed and routed standard cell based design relies on the correctness of guarantees by the standard cell library and PDK designers about timing and power. These might be violated in edge cases not considered. Often used as a last resort, time-consuming, computationally intensive procedure when a fabricated chip design is faulty, if the standard cell library provides the actual geometry, circuit extraction can be performed to recover data for analog SPICE simulation of quantities like fall and rise times and propagation delays to track down the source of the hardware fault. The polygon geometry in the fabrication layers is used to model a section of the chip as discrete devices, with no knowledge of standard cells. It can also be used as a baseline characterization of a standard cell based circuit for developing a custom cell. OpenLane supports the ext2spice circtuit extraction tool provided by the Magic layout viewer [124].

5.9 Design rule checking

The Design Rule Check (DRC) and Electrical Rule Check (ERC) are critical for manufacturability. These rules are specified by the foundry. Design rules are rules about the permissible geometry on each layer and between them. While the most basic rules are easy to describe, like constraints on the metal track width, minimum separation between polygons, or requiring a polygon on the via layer to have metal on both sides, design rules can be scripts with complex formulations by applying rules conditionally. Academic detailed routers like OpenDP rely on DEF [125] files with fairly simple rules. The full complex DRC rules are then checked with an external tool. In OpenLane, this can be either Magic [124] or KLayout [123], the SKY130 open PDK provides DRC rule files for both. Passing DRC relies on the standard cell library being DRC-clean itself and designed with the knowledge about the design rules in the DEF files. ERC defines electrical rules, like disallowing unconnected polygons, shorts, and antenna rules. These need to be evaluated with knowledge about the sources and sinks and helper cells like antenna diodes. Since DRC and ERC make or break the usability of a design flow, in development, the tools used for checking them need to be continually tested with industrial tools [101].

5.10 Design for testability

In the fabrication process of integrated circuits, due to various manufacturing errors, the final manufactured layer geometry can have physical defects of many types. If this defect causes a fault, that is, a modification of the implemented logic in any way, it's possible, but not guaranteed, that it will lead to errors in the behavior of the device. As an example, a dust particle might cause a metal connection to be broken (a defect), leading to a floating input pin of a logic gate (a fault). The output of this logic gate will then no longer fulfill its specification, and cause an error in the computation of the next state of an FSM. This FSM will then get stuck in an intermediate, invalid state, and the device's outputs will no longer fulfill the device behavior specification, possibly causing a failure of the system the device is integrated into.

Defects are inevitable. With large designs on advanced manufacturing processes, a minuscule chance of a defect on a single gate or connection is magnified to a high probability that at least one of the chips on a wafer is faulty. Therefore, integrated circuits must be tested after manufacture ("off-line tests"). Similarly to functional verification of the design, a test is a set of input vectors and expected output vectors. If the outputs of the device match the expected output vectors, the device has passed the test. This is carried out by specialized equipment (automated test equipment, ATE).

Let's first consider testing combinational logic only. Testing all possible input vectors (a "trivial test") is infeasible, since their number is clearly exponential with the number of inputs. In mass production, the time spent on testing each chip adds to its cost. The purpose of a "structural" test is to use knowledge about the circuit structure to find as many faults on each connection between gates as possible. A "functional" (black-box) test has to test all valid situations that can arise when the device is in use, which typically leads to impractically larger tests. To evaluate how many of the possible internal faults a structural test covers, we need a model of these faults. A test covering all modeled faults is a "complete test". The simplest model is the stuck-at model, where each connection can have two faults, where its value is stuck at 0 or at 1. More complex faults, like an input being driven by a different non-constant signal, can be typically detected by stuck-at tests. This testing doesn't take into account faults in signal propagation times or transient faults. Transient faults can be covered by on-line testing while the device is in service.

For a fault to be detected by a test, it must change the output. For example, if we were to test an input of a single OR-gate for being stuck at 0, we would need to set that input to 1 to "excite" the fault, and set the other input to 0 to "propagate" the fault. For larger circuits, this only partially specifies the input vector. This allows us to test multiple faults with a single vector. Some faults may be impossible to observe ("redundant" faults). However, that also guarantees that these will never lead to errors, as long as only one

is present. To make automatic test pattern generation (ATPG) feasible, we only consider each fault in isolation. Notice that synthesis helps eliminate unnecessary logic and reduces the overall computation needed for ATPG. Generally, ATPG works by building a fault list, finding reductions between faults, and iteratively generating test patterns. The reduction step finds which faults "dominate" others. Fault A dominates fault B if any test for B also detects A. If two faults dominate each other, they are "equivalent". Since finding the equivalency classes of a set of faults is NP-complete [126], only "structurally equivalent" faults can be searched in the circuit near each fault. When a test pattern for a fault is generated, it's then simulated to find other faults it also covers, and these are removed from the fault list. This is continued until all faults are covered. Finally, the test can be "statically compacted" by combining their patterns. Two patterns can be combined if their specified inputs don't overlap, for example, 0X and X1 can be combined into 01, where X represents "don't care". Finding these overlaps reduces to the unate covering problem, which is NP-complete [74]. For storage and transfer, the test can also be compressed with classic compression algorithms.

Generating test patterns for each fault can be implemented with algorithms such as PODEM [127] with exponential runtime or its later improvements [128], [129]. Their runtime can be made manageable by limiting the backtracking done while exploring the state space of input vectors, which leads to the algorithm "giving up" on hard to cover faults. ATPG can be also implemented by LEC (see Chapter 3.10) with SAT and some preprocessing [130]. Also, test patterns can be generated pseudo-randomly. As a pseudo-random bit pattern generator, a linear feedback shift register (LFSR) or simple cellular automata can be used. These require minimal ATE storage for the test inputs. This is a "dynamic compaction" method, since the test patterns are generated as the test runs. A combination of either PODEM-based algorithms and SAT ATPG can find complete or near-complete tests at reasonable runtimes.

Testing synchronous designs requires replacing flip flops with scan chain flip flops, which enables serially shifting in arbitrary flip flop states from outside the chip, and reading the values at flip flop inputs.

The unfortunately named Fault [131] is an open source tool for ATPG and scan chain insertion. It performs ATPG with a psuedorandom generator by default, but also supports PODEM and Atalanta [132]. While scan chain flip flops are included in the SKY130 PDK, Fault hasn't been integrated into the OpenLane flow yet, despite documentation suggesting otherwise, and can't be used for SKY130 due to its support for only three types of flip flop.

5.11 Physical design representation

For interoperability of independent or integrated tools in a physical design flow, a shared representation of the intermediate states of the design must be defined and implemented. This representation has the form of a database

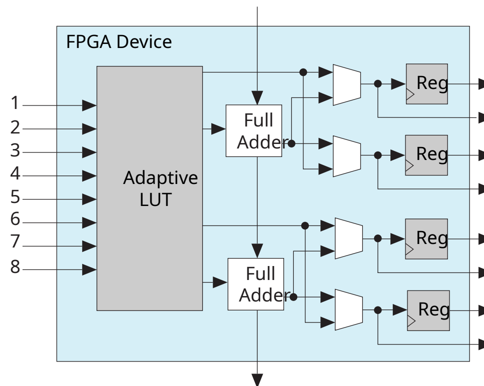


Figure 5.7: Intel Arria 10 BEL, "Adaptive Logic Module", adapted [134]

(DB). Supported types of objects must cover all the information needed to be passed between tools not covered by other standards like SDC for timing or DEF for macros. It can also be advantageous for it to cover the data structures internal to the tools for consistency. For design checkpoints, deterministic partial reruns, and solution space exploration to answer the question "how will this parameter change the final outcome", the DB must be serializable to disk. For retrieval and modification performance, it must be possible to hold it in memory and cache-efficient. To handle real-world designs, it can be required to be only held partially in memory.

LEF (Library Exchange Format) and DEF (Design Exchange Format) [125] file formats are an open standard, originally developed by Cadence and Synopsys. LEF describes the available layers, vias, design rules, cell geometry, and IO pads. DEF describes the geometry and pins of a macro as composed from LEF objects. OpenDB [133], the DB for OpenROAD, is capable of holding and writing the contents of LEF and DEF.

As an example of what detailed routing needs from a DB is shown well in the TritonRoute paper [120].

5.12 FPGA placement and routing

The problems of physical design automation extend to mapping logic to the available resources on FPGAs. As described in Chapter 5.1, any combinational logic can be expressed in a sufficient number of LUTs. In contemporary FPGA architectures, these are multiplexed with registers and arithmetic circuits into "basic elements of logic" (BEL) as illustrated in fig. 5.7.

The FPGA place-and-route toolkit maintained by the creators of Yosys is nextpnr [135]. To prepare the design for nexpnr, the Yosys `abc` pass is used to map combinational logic to generic k-LUTs, where k is specific to the target architecture. These are then directly mapped with `techmap` to target-specific LUT primitives. If supported, DSP blocks with multipliers, block RAMs

(BRAM), or LUTs used to implement RAM (LUTRAM) are also mapped with `techmap` beforehand. I/O pads need to be mapped as well with `iopadmap`.

Nextpnr features two placers: a simple simulated annealing placer, and an analytical placer [136], which adapts Bound2bound-based SimPL [97]. The differences between the problem stem from the different constraints on valid block positions. This is solved by legalizing each type of block separately. Also, design-specific constraints arise as well, like carry chains in BEL adders causing some BELs to be required to be placed in a column, because their signals aren't routable. For routing, two routers are provided, a simple A*-based router, and one based on CRoute [137], which a negotiation-based [118] A* router with a bias to route each source-sink pair closer to the geometric mean of the net to decrease wirelength by sharing tracks. These routers don't have separate global and detailed phases. A timing optimizer [138] to search for improvements to the legal placement is provided, but only used for the Lattice iCE40 FPGA family. By default, the analytical placer is used. The default router varies among supported FPGA architectures.

Once blocks are placed and routed, the design must be converted to a bitstream to configure the FPGA. Bitstream formats and timing models aren't published by vendors[3], so timing characteristics need to be inferred from the behavior of official tooling, and bitstreams need to be reverse engineered by fuzzing [139]. Most supported FPGA architectures remain only partially supported and while the quality of placed and routed designs lags significantly behind official tooling, the runtimes are often far superior. Lattice iCE40 are basically fully supported by the nextpnr flow and the low runtime has already enabled a design that would simply be impossible with official tools, the Glasgow interface explorer [140], a feature-rich logic analyzer with built-in protocol analyzer support which is selected and reconfigured by rebuilding the RTL with the fully open source flow and the bitstream written to the device in seconds.

5.13 Conclusion

OpenROAD is production-ready. It has successfully been used for hundreds of tapeouts on mature nodes as well as some on advanced nodes, such as a 12 nm Global Foundries tapeout [141]. It's composed of leading academic physical design automation tools, many of which have been developed in collaboration with major chip vendors, and strongly integrated with flow scripts for automation, the alternative to which is OpenLane by eFabless. With its permissive open source license, it is attractive for companies to evaluate it for even high performance designs with separately designed mixed-signal macros. It's timing-aware with tunable performance and reproducibility. The

[3]This is due to a combination of FPGA architecture intellectual property protection and, according to an anonymous source, pressure on FPGA vendors by defense customers to prevent reverse engineering and replication of their products.

difference between academic and industrial tools seems to be increasingly limited to the level of utilization of platform and design type specific optimizations. Its runtime goal is a 24-hour full automated chip build with no human-in-the-loop, which it meets for moderately sized designs.

It should be noted that OpenROAD isn't the only or first opensource full chip design flow. QFlow [122] uses Yosys [2] for synthesis, a fork of the simulated annealing-based TimberWolf [96] place-and-route tool, a simple detailed router and STA tool. It integrates with the Magic layout viewer [124] and uses it for circuit extraction and GDS generation. It also features the netgen LVS tool, also used by OpenLane. The maintainer of QFlow is now involved in the development of both OpenROAD and OpenLane [142].

Coriolis [143] uses a parallel Bound2bound placer with a conjugate gradient solver [144]. This global placer, based on SimPL [97], determines the density penalty between wirelength optimization iterations by reformulating the density reduction as a transportation problem [145]. placing still look for improvements to the wirelength, unlike OpenROAD's "dead simple" and correspondingly fast detailed placer, which relies more on the global placement quality. The Katana global router uses a heuristic Dijkstra search with a distance function penalized for congestion and vias, rather than constructing rectilinear Steiner trees. The Kite detailed router is a negotiation-based R&R router. Coriolis has inherited tooling from its predecessor, the Alliance flow, specifically, the HiTas STA engine, cougar circuit extractor, lvx LVS tool, and DRuC DRC tool. Coriolis is under active development at the time of writing.

Both QFlow and Coriolis explicitly state that the user should not expect them to provide good results in large designs and recent nodes. Neither is aware of the slack available in nets when placing and are built on design decisions that might not scale well. QFlow using a maze router for the entire chip implies it's more likely to expend large amounts of computational effort on ripping up and rerouting connections to avoid congested areas, rather than guiding the maze routing away in the first place with a global router.

Interoperability is limited by the integration of each flow with its own DB. Additionally, most historic academic publications have stayed closed source and designed only to operate on academic benchmarking files. Some software for interoperability between open, academic, and industrial tools has been developed with various design goals [146]–[148].

Appendix A

Experiments

A.1 Yosys temporal logic counterexample loop detection

The Yosys tool chain supports formal verification flows through the SMT2 backend. This backend converts the internal representation of a circuit (RTLIL), including formal assertions, assumptions, and coverage statements, into a set of SMT-LIB v2 clauses, which pose the question of property violation satisfiability. Afterwards, an SMT solver is automatically invoked. If the clause set is satisfiable, the solver is extracted to produce a counterexample. This counterexample is composed of sequences of values of module ports and signals defined as `$anyseq` or `$anyconst`. It's then dumped as an SMT-LIB v2 file, as a Verilog testbench to actually drive ports and underspecified signals, and a VCD file containing the trace of the design running in time with these signal values.

This process initially wasn't in any way detecting a loop in the counterexample. I have experimentally implemented it as a pull request, which has been accepted [149]. The code changes are limited to `smtbmc.py`, a Python script for launching solvers with correct arguments and post-processing their results into VCD and Verilog. Counterexample loop detection is now available under the `--detect-loops` flag for the `yosys-smtbmc` command and can be easily added into Symbiyosys workflows. Symbiyosys is a wrapper on top of yosys to easily declare HDL files and formal methods. Since the time step in temporal induction doesn't necessarily correspond to a single concrete clock due to the required `clk2fflogic` pass preceding formal passes, it is simply called a "step". It's the number of these steps that the user specifies as the desired induction length. A list (as long as the induction length) of counterexample signal value lists is obtained and iterated over. A Python dictionary is a complex structure with a simple interface: any hashable (primitive and fixed length, trivially composed of such, or manually supplied with a hash method) object can be used as a dictionary key. For this reason, the signal value list must be converted into a tuple, which is hashable. The dictionary then does

fast insertions and lookups as it's a generic hash table. To detect duplicate steps, a set object would suffice. However, it's clearly helpful to inform the developer between which two steps does the counterexample trace hold identical state vectors. For this reason, for each step, the signal value list is obtained, the `states` dictionary is searched for it (in constant time). If it can't be found, it means no loop has been found yet, and it's associated in the dictionary with the step index. When it's found in the dictionary, the previous associated step index is retrieved, and presented to the user together with the current index. These indices show the first pair of identical states in the counterexample to the user. Initially, only regular signals were considered, and memories were not. This is because memories are often larger than the number of signals in a design and therefore can greatly increase the signal list size and loop detection run time. However, it was straightforward to add memory, and eliminate the false positives that ignoring it can cause.

```
Trying induction in step 1..
Trying induction in step 0..
Temporal induction failed!
Assert failed in demo1_mine: demo1_mine.v:13.31-15.32
($assert$demo1_mine.v:13$17)
Writing trace to VCD file: demo1_mine.vcd
Checking for loops in found induction counter example
This feature is experimental and incomplete
Loop detected, increasing induction depth will not help.
Step 2 = step 0
Status: FAILED
```

Listing 4: Temporal induction counterexample loop detection output, simplified

A.2 GHDL PSL bug

As a part of my preparation of a formal verification setup for the CTU CAN FD IP core [150], I found a bug in GHDL, where unsupported PSL-defined verification declarations would cause a crash, rather than being handled and reported as an error. I reported this bug, which was promptly fixed by the maintainer [151].

A.3 Amaranth combinational loop detection

Amaranth HDL doesn't have a specification, but it has documentation for its single implementation. Here, the user is warned:

> The current version of Amaranth does not detect combinatorial feedback loops, but processes the design under the assumption that

> there aren't any. If the design does in fact contain a combinatorial feedback loop, it will likely be silently miscompiled, though some cases will be detected during synthesis or place & route. This hazard will be eliminated in the future.

I have joined the discussion around this and proposed a solution of the primitive form presented in lst. 5 [152].

```
for every combinational assignment s.eq(e):
    for every signal s' in expression e:
        add edge (s', s)
```

Listing 5: Combinational dependency draph building pseudocode

This requires collecting every combination assignment, which isn't done until the generate step, where either a Python simulation program or yosys internal representation is generated. All operations preceding generation are implemented as tree traversal, modifying the existing design hierarchy, dealing with clock domains, rewriting clock signals to regular signals, and so on. None of them require a flattened view of all assignments in the design. For this reason, I built a tree traversal named `CombGraphCompiler`, which collects assignments as it descends the module (specifically, `Fragment`) hierarchy, runs my `AssignmentGraphBuilder` on each statement in the combinational domain of each `Fragment`, collects their state—an assignment edge list—and returns it. It needs to keep a map from `Signal`s to hashable integer IDs, pass it to `AssignmentGraphBuilder` calls, and take the modified version to pass to the next one. It's also extracted after the `CombGraphCompiler` finishes in order to label a possibly found combinational loop with signal names for error reporting. `AssignmentGraphBuilder` uses existing methods on assignment right-hand side expressions (`Value`s) to extract all signals referenced in them. However, in general, any assignment in a conditional block is also dependent on all condition signals, so these are pushed onto a stack when a `Switch` statement is encountered as the `Value` tree is traversed. When this is finished, we have a graph of all combinational dependencies between signals. After `CombGraphCompiler` is done, the found edges are depth-first searched from each unvisited state. If the design is legal, the graph is a forest, and no cycles will be found. When a cycle is found, an error is returned, and an error is printed as shown in lst. 6.

```
amaranth.hdl.cd.DomainError: Combinational loop detected: a,a,o,o
```

Listing 6: Combinational loop detection output

Adding hierarchically fully qualified names was omitted, as this was a prototype. It worked well, but 11 language unit tests would fail. I traced this down to the implementation of a gray decoder in the Amaranth standard library, shown in lst. 7.

```
class GrayDecoder(Elaboratable):
    def elaborate(self, platform):
        m = Module()
        m.d.comb += self.o[-1].eq(self.i[-1])
        for i in reversed(range(self.width - 1)):
            m.d.comb += self.o[i].eq(self.o[i + 1] ^ self.i[i])
        return m
```

Listing 7: Old GrayDecoder implementation

This design is correct when synthesized, as no bit of `self.o` depends combinationally on itself, but each bit depends on the next, except for the most significant bit. The loop detection creates a false positive, because my pass ignores slices, and treats them as references to the entire signal. To be compatible with this use case, each bit of every signal would have to be considered in isolation throughout this pass, increasing the complexity greatly, and requiring special handling of arithmetical operations, while overapproximating their bit dependencies just to keep the existing functionality, implying an N-bit integer addition would add N squared edges to my graph instead of one. This could be done iteratively only once a possible combinational loop is found, but many such loops could possibly be found, freezing the elaboration process.

Since `GrayDecoder` was the only standard library part using this possibly illegal "hack", I have rewritten it to make the output dependent only on inputs, which has been accepted [153]. Since this module is formally verified against the untouched `GrayEncoder` implementation, unit tests proved the correctness of this change.

```
def elaborate(self, platform):
    m = Module()
    rhs = Const(0)
    for i in reversed(range(self.width)):
        rhs = rhs ^ self.i[i]
        m.d.comb += self.o[i].eq(rhs)
    return m
```

Listing 8: New GrayDecoder implementation

Several large real-world Amaranth projects (RISC-V CPUs, SoC libraries) have been tested with my detection. No false positives were found and slowdown coming from this check is lower than 15% if measurable.

A.4 Minor Yosys bug fixes

Support for the MathSAT solver [65] was broken. The code responsible for parsing the solver standard output, had to be modified, which fixed it, and introduced no change to the functionality for other solvers. I submitted a pull request, which is still open at the time of writing [154].

A faulty design of mine was asking yosys to synthesize memory with word width of 0. Memory size in words was being calculated as `GetSize(init.data) / width` in `kernel/mem.cc`. I added an assertion statement into the `check` method already invoked at the start of the offending method `Mem::emit` to detect this situation and provide clear information how this I submitted a pull request, which was and accepted and merged [155].

A.5 OpenROAD

A.5.1 CTU CAN FD IP Core tape-in

The CTU CAN FD IP core [156] is an implementation of the CAN bus with support for the Flexible Data-rate extension. The core has been developed at the Department of Measurement at FEE CTU. I have decided to use it as a case study for the OpenLane flow with OpenROAD physical synthesis. After synthesis, its size is approximately 18,000 cells, about half of which are multiplexers, see listing 9. Buffer depths were left at their minimal values by defaults. In a real tapeout, these buffers should be implemented with proper SRAM cells to save die area as illustrated in fig. A.1. SRAM cells are included in the SKY130 open PDK, but implementing SRAM also requires controllers, address decoders, drivers, precharge circuitry, and sense amplifiers. This is covered by the OpenRAM [157] project, provides an open tool for generating these parametrically. OpenRAM is currently only experimentally available for SKY130. Original plans to include an SRAM memory in the eFabless-designed Caravel harness that all OpenMPW projects are encapsulated in are paused, replaced with flip flop based memory.

The physical design process by OpenROAD is illustrated in fig. A.2, fig. A.3, and fig. A.4 using scripts I published on GitHub, including the design sources [158], [159].

Since the core is designed with VHDL, I used the GHDL [29] compiler and simulator as integrated into a plugin for Yosys [160] to convert it to Verilog with the Yosys `write_verilog` command in order to build the design with OpenLane.

CAN bus is standardized as ISO 11898, CAN FD as ISO11898-1:2015. They are also published with free access. However, the CAN protocols are all

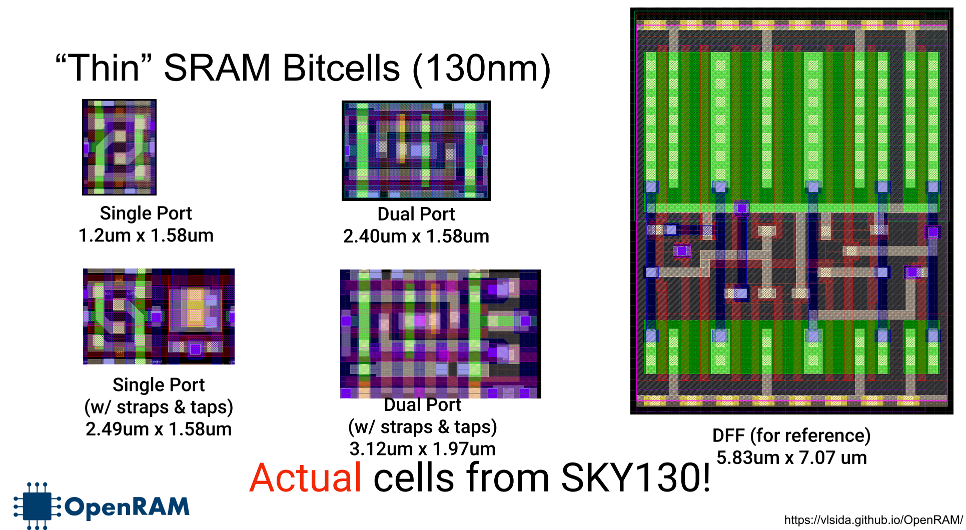


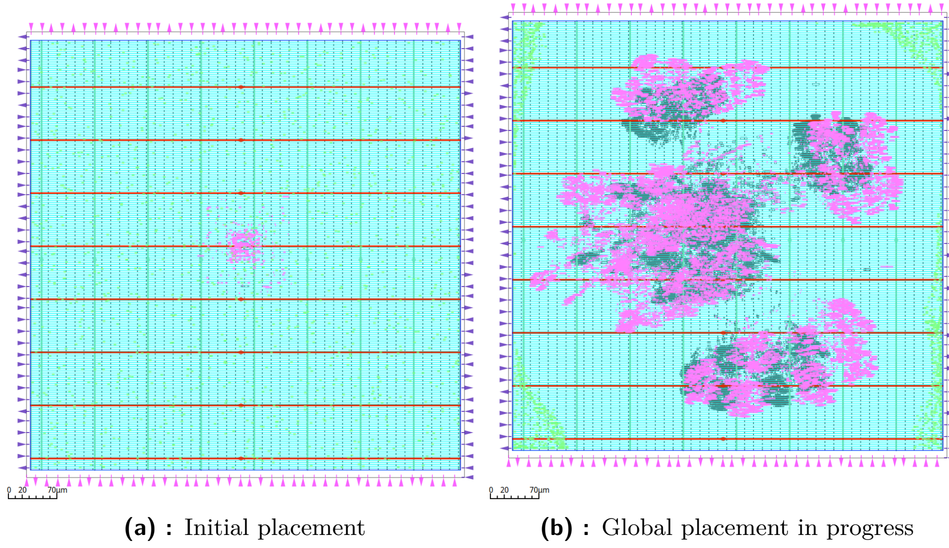
Figure A.1: Cell size comparison, adapted [157]

patented by Robert Bosch GmbH. For this reason, I have not finalized the design for Google-sponsored OpenMPW tapeout due to the requirement of purchasing a CAN bus implementation license.

I also tested ATPG with Fault [131] on this design. Since SKY130 is not supported, I use the OSU 35nm cell library it comes with. ATPG took 37 minutes on a 6-core Ryzen 3600 desktop CPU, exceeding 20GB of RAM usage. The ATPG outputs are a `.svf` file under 1MB containing the test patterns, and 326MB of JSON I/O metadata.

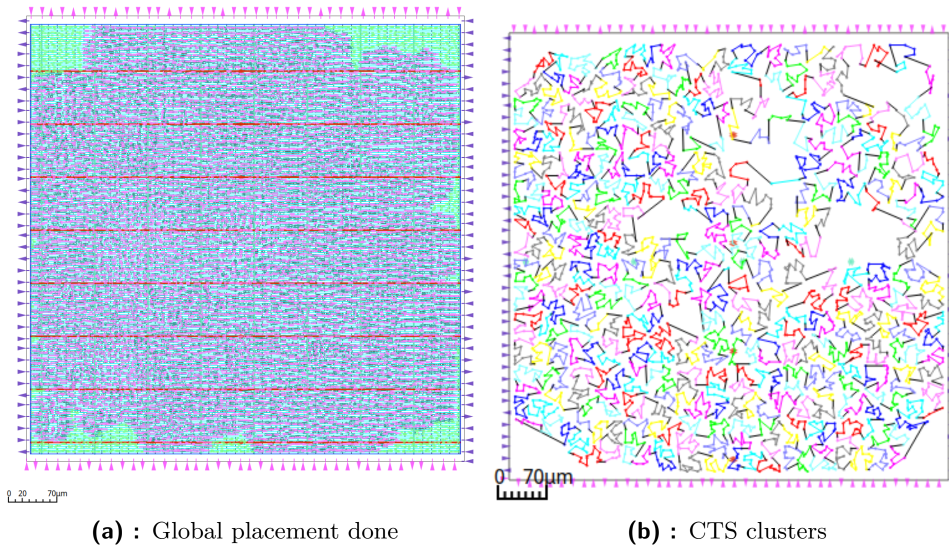
```
Number of cells:              17547
    $_ANDNOT_                    2627
    $_AND_                        266
    $_MUX_                       6745
    $_NAND_                       580
    $_NOR_                        426
    $_NOT_                        488
    $_ORNOT_                      459
    $_OR_                        1760
    $_XNOR_                       104
    $_XOR_                        611
    sky130_fd_sc_hd__dfrtp_2      990
    sky130_fd_sc_hd__dfstp_2       91
    sky130_fd_sc_hd__dfxtp_2     2400
```

Listing 9: CTU CAN FD Yosys synthesis cell number report



(a) : Initial placement **(b) :** Global placement in progress

Figure A.2: CTU CAN FD IP core physical design



(a) : Global placement done **(b) :** CTS clusters

Figure A.3: CTU CAN FD IP core physical design

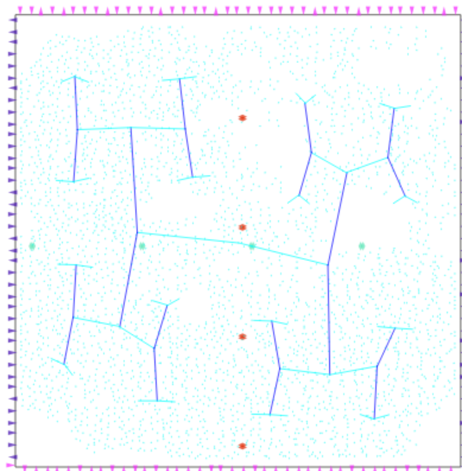


Figure A.4: CTU CAN FD IP core clock tree synthesis H-tree

A.5.2 TinyTapeout

As a means of enabling hobbyists and students to create their own silicon, the TinyTapeout [161] project aims to implement a multi-project chip with more than a hundred individual designs each on the order of only hundreds of gates. The inputs and outputs of the designs are connected in a scan-chain, limiting the maximum frequency to approximately 20kHz.

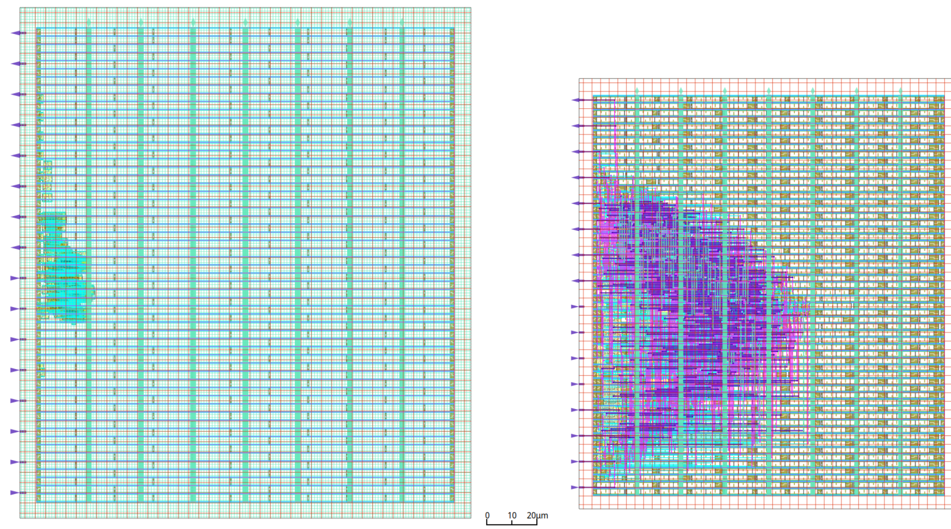
I have joined the second run with an Amaranth design with two primitive functionalities, selected by an input pin. The design is publicly available on GitHub [162].

- Chase the beat: An output (with a presumably connected LED) is always high. The time delta between two taps of an input button set the period at which it's changed which output is high.
- Noise generator: All eight outputs are driven each with a linear feedback shift register output. The outputs should be connected to buffers and an R-2R ladder for digital to analog conversion. This should create approximately white audio noise.

The physical design process by OpenROAD is illustrated in fig. A.6 and fig. A.7.

A shared physical design configuration was initially meant to be used for all designs. This included `PL_BASIC_PLACEMENT` to be set to 1, which is a flag for OpenLane to set options for the OpenROAD global placer to terminate with 90% acceptable overflow (a measure of cell overlap) and limit the placer iterations to 20. This was deemed necessary for most extremely small designs (under about 50 gates). However, some users with larger designs discovered that removing this flag drastically cut the runtime and resulting area or removed routing failure. Also, the designs built with the default configuration had suspicious triangle shapes, with logic cells only at the edge, as shown in fig. A.5b. I investigated this problem.

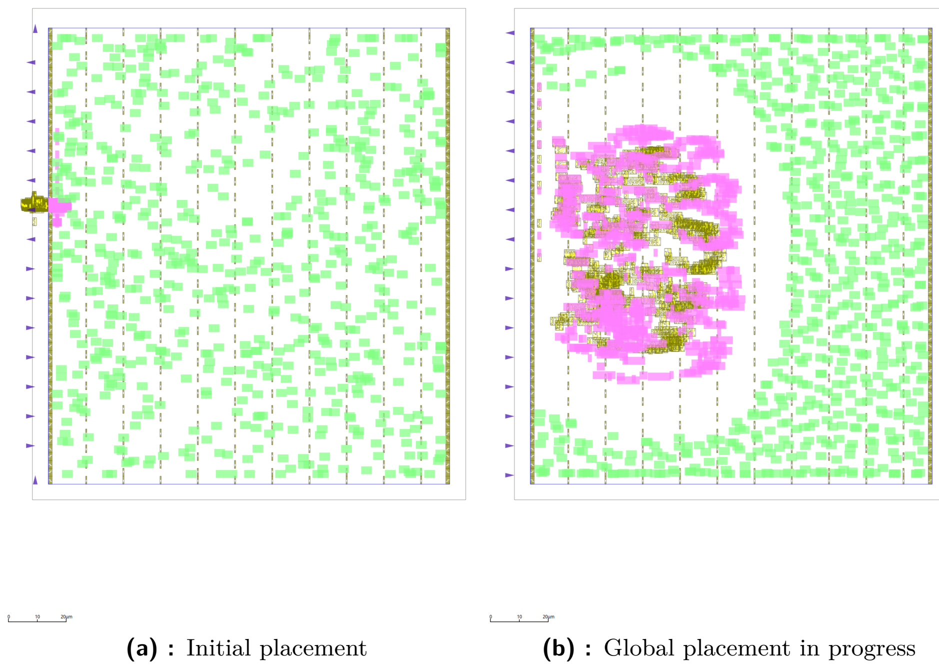
Without the flag, extremely small designs such as the Verilog example repository would fail in the placement step with the error `RePlAce divergence detected. Re-run with a smaller max_phi_cof value`. The flag was preventing the placer to run for sufficient number of iterations for the solver to diverge, allowing the flow to typically succeed, but for example my design required almost 500 iterations to terminate without the flag. In fact, the global placer was terminating with all cells almost in the same place as after the initial placement, localized to a small area (see fig. A.5a). This had a profound impact on the rest of the flow. The detailed placer in OpenROAD is simple and only searches for closest valid space to place cells to. The global placer doesn't just place the logic cells but also virtual filler cells to make the design well packed. In a proper global placement run, the filler cells would be pushed away from the logic, since the logic is optimized for wirelength and the solver keeps connected logic



(a) : Overflowing global placement result **(b) :** Routed design

Figure A.5: TinyTapeout physical design, with `PL_BASIC_PLACEMENT`

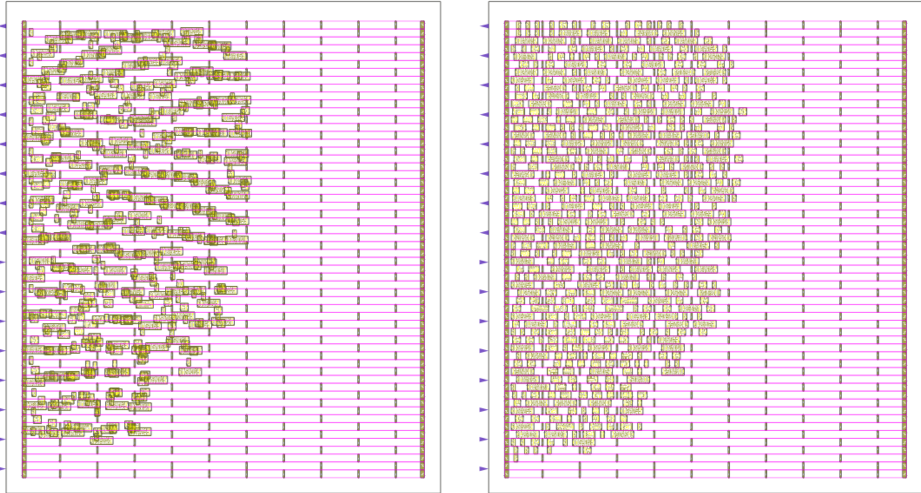
cells together. However, the detailed placer would in this case receive a design with logic and filler cells all in almost the same place. It would then move all these cells to the nearest (in the Manhattan distance sense) valid coordinates, which would create a diamond shape if it wasn't for the I/O pins causing the initial placement to be near the left edge of the design. The global and detailed router would then receive logic cells placed in a V shape with connected cells not being as close as possible. This would lead to long router runtimes (with up to hours-long hangs in the GDS-building GitHub Action). For some designs the routing capacity just wasn't sufficient, causing the flow to fail. The error suggests lowering `max_phi_cof`, which forces smaller iteration steps, preventing solver divergence. However, a very small one had to be chosen, greatly increasing the number of iterations required, even for the tiny designs. OpenLane does have an option to cope with solver divergence on tiny designs, `PL_RANDOM_INITIAL_PLACEMENT`, but the random generator Python script doesn't take a seed from the config file, so it can't be used for any actual tapeouts where deterministic builds are obviously required. I have submitted an issue for all randomness in OpenLane to take a seed [163]. I have captured an animation of the internal state of the global placer as an animation, which clearly displays the diverging behavior, which is included in the attachments.



(a) : Initial placement

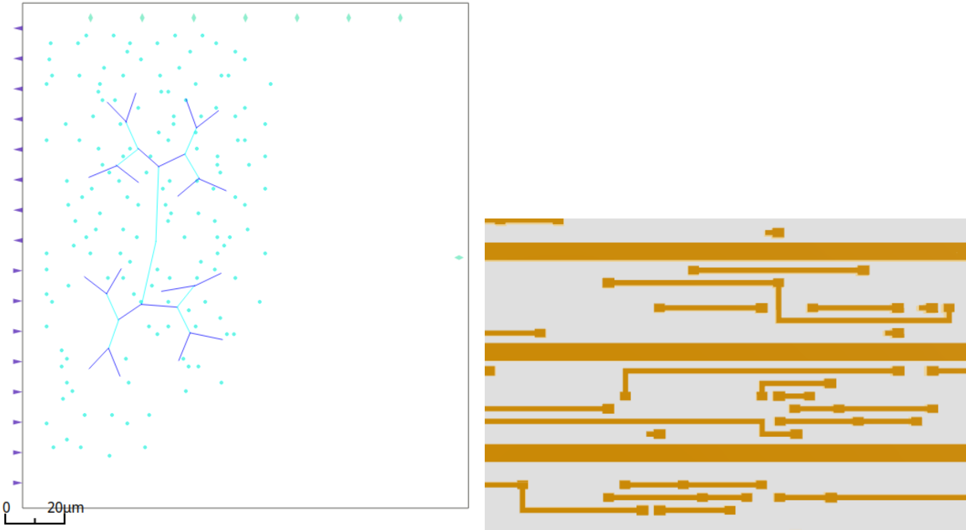
(b) : Global placement in progress

Figure A.6: TinyTapeout physical design



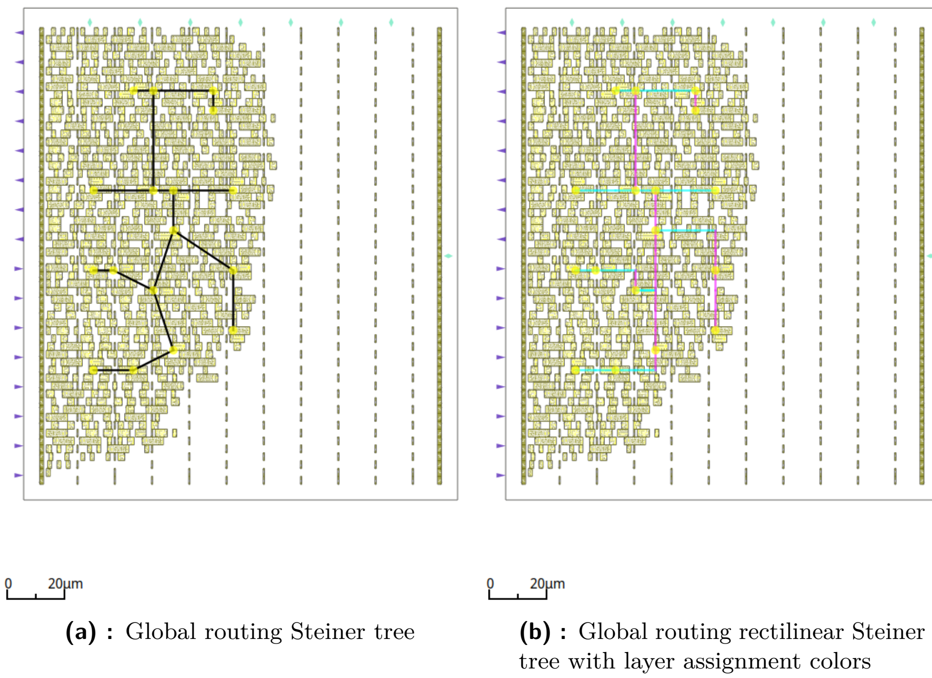
(a) : Global placement done **(b) :** Detailed placement done

Figure A.7: TinyTapeout physical design



(a) : CTS H-tree **(b) :** Mixed direction routing on the first metal layer

Figure A.8: TinyTapeout physical design



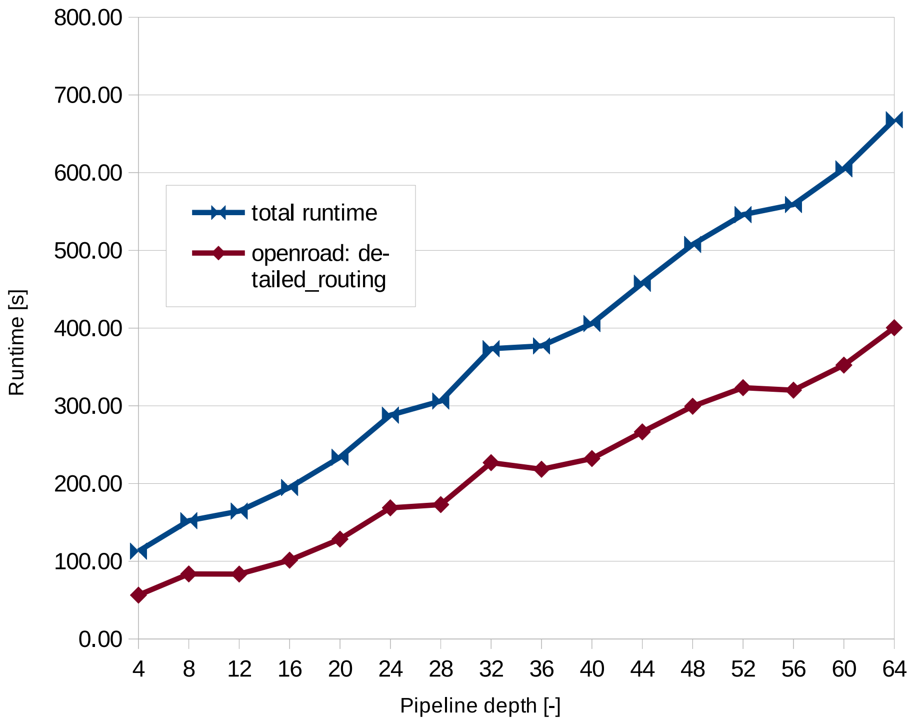
(a) : Global routing Steiner tree

(b) : Global routing rectilinear Steiner tree with layer assignment colors

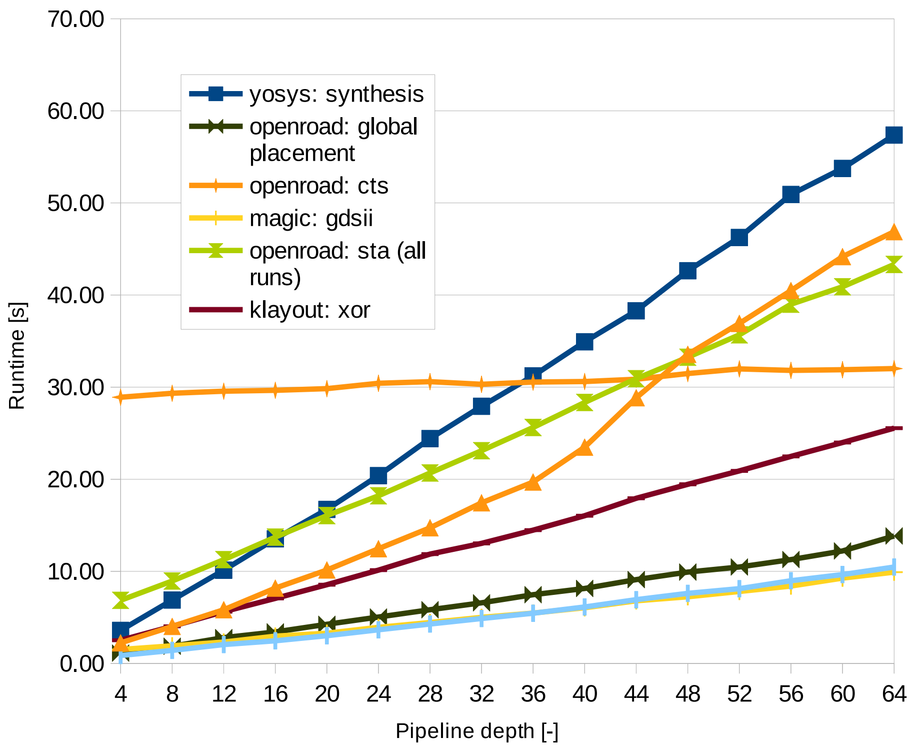
Figure A.9: TinyTapeout physical design

A.5.3 Benchmark

OpenROAD presentations declare fast design iterations as a core project principle. To see how steps in the flow contribute to this outcome, I have designed a benchmark circuit in Amaranth HDL. It's a parametrized pipeline with a variable bit width and number of stages. The inputs to each stage are constructed each as an AND-inverter graph. Whether each edge is complemented is random, with a fixed seed. Increasing the pipeline depth is an easy way of linearly adding more logic without synthesis opportunities to merge subgraphs between stages. Increasing the width also increases the amount of logic linearly. I ran a sweep across both of these parameters separately and plotted the contributions from elements in the OpenLane flow. Very fast tasks are excluded from the detailed charts. The results are presented in fig. A.10 and fig. A.11. It is apparent that the total runtime is even for this small, unconstrained design dominated by the detailed routing phase. This can be expected, since path finding is a task with no good shortcuts. A moderate target cell density of 55% was chosen as a typical target among OpenLane examples. The constant time spent on clock tree synthesis is something I didn't find a satisfying explanation, profiling the tool would be necessary. Most tasks seem to scale approximately linearly. The depths and widths were limited by the time required to build the Amaranth design and emit Verilog.

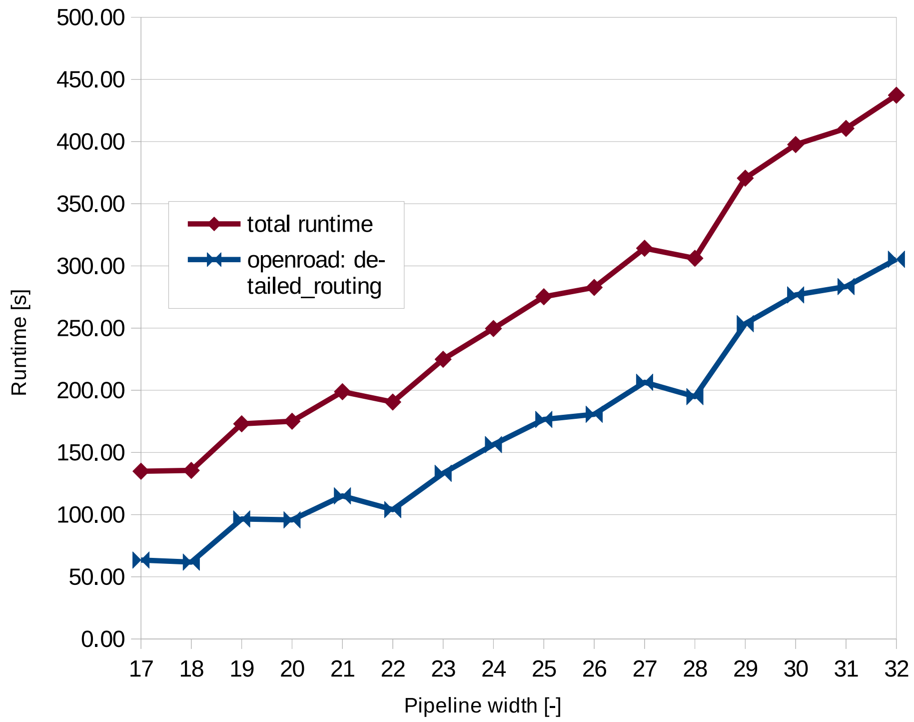


(a) : Runtime breakdown

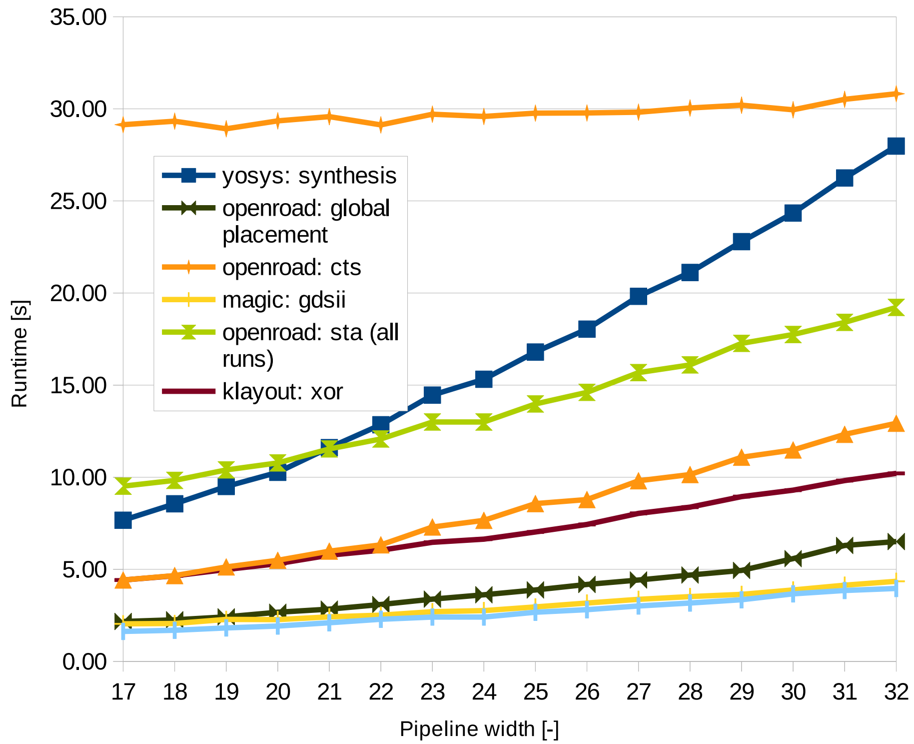


(b) : Excluding detailed routing

Figure A.10: Pipeline depth sweep



(a) : Runtime breakdown



(b) : Excluding detailed routing

Figure A.11: Pipeline width sweep

Appendix B

Attachments

The attachments contains animations extracted from the OpenROAD global placer.

- `chase.webm`: successful placement of my TinyTapeout design.
- `ctu-can-fd.webm`: successful placement of the CTU CAN FD IP core. It is notable that as a low-overlap solution is found, the state reverts to an earlier snapshot in a backtracking attempt to improve routability.
- `demo-diverge.webm`: failing placement of the TinyTapeout Verilog example project. The solver divergence manifests itself with a back-and-forth movement and filler cells mixing with the logic cells.

I bulit everything else publicly at the respective git repositories. Specifically:

- TinyTapeout 02 submission [162]
- OpenROAD modifications needed for visualization I presented [164]
- CTU CAN FD translated Verilog and OpenLane modifications needed for visualization and benchmarking I presented [158]
- Additional scripts for benchmarking and visualization [159]

Appendix C

Literature sources

For further explanations of concepts not cited directly, I recommend the resources presented here. Specifically:

- General statements about logic synthesis problems and logic representations in “Logic synthesis and verification algorithms” [165]
- General statements about properties and model checking in Chapter 3 in “Principles of model checking” [55]
- Constructing propositions to model and verify systems as well as descriptions of SAT and SMT solvers in “Decision procedures: an algorithmic point of view” [56]
- Detailed and formal definitions and constructions for model checking in “Model Checking” course lectures at RWTH Aachen [166]
- Examples of select physical design algorithm up to 2008 in “Practical problems in VLSI physical design automation” [92]
- A wider and deeper tour through physical design algorithms up to 2011 in “VLSI physical design: from graph partitioning to timing closure” [167]
- General statements about testing and design for testability in Chapter 5.10 in “VLSI test principles and architectures: design for testability” [168]

Appendix D

Bibliography

[1] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International conference on computer aided verification*, 2010, pp. 24–40.

[2] C. Wolf, "Yosys open SYnthesis suite manual." Available: `https://yosyshq.readthedocs.io/projects/yosys/en/latest/`

[3] "CologneChip GateMate FPGA toolchain installation user guide." CologneChip, 2022. Available: `https://www.colognechip.com/docs/ug1002-toolchain-install-latest.pdf`

[4] J.-L. Aufranc, "Renesas introduces sub 50 cents FPGA family with free yosys-based development tools." CNX Software, 2021. Available: `https://www.cnx-software.com/2021/11/22/renesas-50-cents-fpga-forgefpga-yosys-development-tools/`

[5] T. Ajayi and D. Blaauw, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," in *Proceedings of government microcircuit applications and critical technology conference*, 2019.

[6] Y.-M. Yang and I. H.-R. Jiang, "Analog placement and global routing considering wiring symmetry," in *2010 11th international symposium on quality electronic design (ISQED)*, 2010, pp. 618–623.

[7] "Xilinx FFT IP core." 2022. Available: `https://www.xilinx.com/products/intellectual-property/fft.html`

[8] "IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows," *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009)*, pp. 1–510, 2014, doi: 10.1109/IEEESTD.2014.6898803.

[9] A. Kamppi *et al.*, "Kactus2: A graphical EDA tool built on the IP-XACT standard," *Journal of Open Source Software*, vol. 2, no. 13, p. 151, 2017.

[10] O. Kindgren, "Ipyxact, a python-based IP-XACT parser." 2023. Available: `https://github.com/olofk/ipyxact`

[11] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann, "LiteX: An open-source SoC builder and library based on migen python DSL," in *OSDA 2019, colocated with DATE 2019 design automation and test in europe*, 2019.

[12] O. Kindgren, "FuseSoC, a hardware package manager." 2023. Available: `https://github.com/olofk/fusesoc`

[13] O. Kindgren, "Edalize, a python library for interacting with EDA tools." 2023. Available: `https://github.com/olofk/edalize`

[14] "IEEE 754 floating point unit in verilog." 2016. Available: `https://github.com/danshanley/FPU`

[15] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the FIRRTL language," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9*, 2016.

[16] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC design automation conference 2012*, 2012, pp. 1212–1221.

[17] C. Lattner *et al.*, "MLIR: A compiler infrastructure for the end of moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[18] "CIRCT: Circuit IR compilers and tools." LLVM Project, 2023. Available: `https://circt.llvm.org/`

[19] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: A multi-level intermediate representation for hardware description languages," in *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, 2020, pp. 258–271.

[20] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, 2021, pp. 804–817.

[21] H. Verilog, "Language reference manual." IEEE, 2001.

[22] M.-K. YOU and G.-Y. SONG, "SystemVerilog 3.1 a language reference manual: Accellera's extensions to verilog SystemVerilog 3.1 a language reference manual: Accellera's extensions to verilog, 2004," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 93, no. 5, pp. 989–992, 2010.

[23] M. Popoloski, "Slang - SystemVerilog language services." 2023. Available: `https://github.com/MikePopoloski/slang`

[24] C. Alliance, "SV-tests: Test suite to check SystemVerilog compliance." 2023. Available: `https://chipsalliance.github.io/sv-tests-results/`

[25] S. Hoover and A. Salman, "Top-down transaction-level design with TL-verilog," *arXiv preprint arXiv:1811.01780*, 2018.

[26] "Silice: A language for hardcoding algorithms into FPGA hardware." 2023. Available: https://github.com/sylefeb/Silice

[27] I. D. A. S. Committee *et al.*, "„Std 1076–2008, IEEE standard VHDL language reference manual "," *IEEE, New York, NY, USA*, 2008.

[28] J. Decaluwe, "VHDL's crown jewel." 2010. Available: https://insights.sigasi.com/opinion/jan/vhdls-crown-jewel/

[29] T. Gingold, "GHDL, a VHDL compiler." 2002. Available: http://ghdl.free.fr/ghdl

[30] "Spinal hardware description language documentation." 2023. Available: https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html

[31] W. Snyder, "Verilator: Speedy reference models, direct from RTL," *Presentation to University of Massachusetts Amherst*, 2017.

[32] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "Cλash: Structural descriptions of synchronous hardware using haskell," in *2010 13th euromicro conference on digital system design: Architectures, methods and tools*, 2010, pp. 714–721.

[33] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in *High-level synthesis*, Springer, 2008, pp. 129–146.

[34] T. Bourgeat, C. Pit-Claudel, and A. Chlipala, "The essence of bluespec: A core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, 2020, pp. 243–257.

[35] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, *et al.*, "Kami: A platform for high-level parametric hardware specification and its modular verification," 2017.

[36] "The coq proof assistant 8.16.1 reference manual." 2022. Available: https://coq.inria.fr/distrib/current/refman/

[37] M. Käyrä and T. D. Hämäläinen, "A survey on system-on-a-chip design using chisel HW construction language," in *IECON 2021–47th annual conference of the IEEE industrial electronics society*, 2021, pp. 1–6.

[38] G. Hempel, "Generation of application specific hardware extensions for hybrid architectures: The development of PIRANHA-a GCC plugin for high-level-synthesis," PhD thesis, Dissertation, Dresden, Technische Universität Dresden, 2018, 2019.

[39] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[40] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation: What does the NP-completeness proof of chaitin et al. Really prove? Or revisiting register allocation: Why and how," in *International workshop on languages and compilers for parallel computing*, 2007, pp. 283–298.

[41] Y. Mengmeng and L. Jie, "Register allocation based on boolean satisfiability," in *Intelligent data analysis and its applications, volume II*, Springer, 2014, pp. 275–281.

[42] D. Das, S. A. Ahmad, and K. Venkataramanan, "Deep learning-based hybrid graph-coloring algorithm for register allocation," *arXiv preprint arXiv:1912.03700*, 2019.

[43] "Polly: LLVM framework for high-level loop and data-locality optimizations." LLVM project, 2023. Available: `https://polly.llvm.org/`

[44] "Auto-vectorization in LLVM." LLVM project, 2023. Available: `https://llvm.org/docs/Vectorizers.html`

[45] Aki-nyan, "Fast parallel sliding-window based binary diff." 2020. Available: `https://lethalbit.net/blog/fast-diff/`

[46] "XLS: Accelerated HW synthesis, scheduling overview." Google, 2023. Available: `https://google.github.io/xls/scheduling/pipeline-scheduling`

[47] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *International conference on computer aided verification*, 1996, pp. 419–422.

[48] A. V. Yakovlev and A. M. Koelmans, "Petri nets and digital hardware design," in *Advanced course on petri nets*, 1996, pp. 154–236.

[49] J. Strejcek, "Linear temporal logic: Expressiveness and model checking," PhD thesis, PhD thesis, Faculty of Informatics, Masaryk University in Brno, 2004.

[50] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on logic of programs*, 1981, pp. 52–71.

[51] E. A. Emerson and J. Y. Halpern, "'Sometimes' and 'not never' revisited: On branching versus linear time temporal logic," *Journal of the ACM (JACM)*, vol. 33, no. 1, pp. 151–178, 1986.

[52] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking." *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.

[53] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *International conference on formal methods in computer-aided design*, 2000, pp. 127–144.

[54] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[55] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[56] O. Strichman, *Decision procedures: An algorithmic point of view*. Springer, 2008.

[57] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[58] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM (JACM)*, vol. 7, no. 3, pp. 201–215, 1960.

[59] "Formally verified SAT-solver IsaSAT wins EDA challenge," 2021, Available: `https://www.utwente.nl/en/eemcs/fmt/news-events/news/2021/7/1124102/formally-verified-sat-solver-isasat-wins-eda-challenge`

[60] "SAT competition 2022 results." 2022. Available: `https://satcompetition.github.io/2022/results.html`

[61] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Rtlv: Push-button verification of software on hardware," Available: `https://people.csail.mit.edu/nickolai/papers/moroze-rtlv.pdf`

[62] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *International conference on tools and algorithms for the construction and analysis of systems*, 2009, pp. 174–177.

[63] A. Niemetz, "Bit-precise reasoning beyond bit-blasting/eingereicht von dipl.-ing. Aina niemetz, bsc," PhD thesis, Universität Linz, 2017.

[64] C. W. Barrett *et al.*, "CVC4," in *Computer aided verification - 23rd international conference, CAV 2011, snowbird, UT, USA, july 14-20, 2011. proceedings*, 2011, vol. 6806, pp. 171–177. doi: 10.1007/978-3-642-22110-1_14.

[65] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The mathsat 4 smt solver," in *International conference on computer aided verification*, 2008, pp. 299–303.

[66] B. Dutertre, "Yices 2.2," in *Computer-aided verification (CAV'2014)*, 2014, vol. 8559, pp. 737–744.

[67] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on tools and algorithms for the construction and analysis of systems*, 2008, pp. 337–340.

[68] C. Wolf, "SymbiYosys," *URL: https://symbiyosys. readthedocs. io/.[Cited on page 6.]*.

[69] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *International conference on automated deduction*, 2015, pp. 378–388.

[70] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas, "Effective theorem proving for hardware verification," in *International conference on theorem provers in circuit design*, 1994, pp. 203–222.

[71] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, J. H. Siekmann and G. Wrightson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. doi: 10.1007/978-3-642-81955-1_28.

[72] "Cadence JasperGold." 2022. Available: `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html`

[73] "Siemens questa." 2022. Available: `https://eda.sw.siemens.com/en-US/ic/questa/`

[74] O. Coudert, "On solving covering problems," in *Proceedings of the 33rd annual design automation conference*, 1996, pp. 197–202.

[75] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*, vol. 2. Springer Science & Business Media, 1984.

[76] P. Fišer and H. Kubátová, "Two-level boolean minimizer boom-ii," in *Proc. 6th int. Workshop on boolean problems (IWSBP'04), freiberg, germany*, 2004, vol. 23.

[77] P. Fiser and D. Toman, "A fast SOP minimizer for logic funcions described by many product terms," in *2009 12th euromicro conference on digital system design, architectures, methods and tools*, 2009, pp. 757–764.

[78] N. Song and M. A. Perkowski, "EXORCISM-MV-2: Minimization of exclusive sum of products expressions for multiple-valued input incompletely specified functions," in *[1993] proceedings of the twenty-third international symposium on multiple-valued logic*, 1993, pp. 132–137.

[79] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Transactions on computers*, vol. 45, no. 9, pp. 993–1002, 1996.

[80] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," ERL Technical Report, 2005.

[81] F. Mailhot and G. De Micheli, "Technology mapping using boolean matching and don't care sets." in *EURO-DAC*, 1990, vol. 90, pp. 212–216.

[82] R. K. Brayton, "Compatible observability don't cares revisited," in *IEEE/ACM international conference on computer aided design. ICCAD 2001. IEEE/ACM digest of technical papers (cat. No. 01CH37281)*, 2001, pp. 618–623.

[83] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," in *Theory of machines and computations*, Elsevier, 1971, pp. 189–196.

[84] A. Nerode, “Linear automaton transformations,” *Proceedings of the American Mathematical Society*, vol. 9, no. 4, pp. 541–544, 1958.

[85] J.-K. Rho, G. D. Hachtel, F. Somenzi, and R. M. Jacoby, “Exact and heuristic algorithms for the minimization of incompletely specified state machines,” *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 13, no. 2, pp. 167–177, 1994.

[86] C. E. Leiserson and J. B. Saxe, “Retiming synchronous circuitry,” *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[87] P. Pan, “Continuous retiming: Algorithms and applications,” in *Proceedings international conference on computer design VLSI in computers and processors*, 1997, pp. 116–121.

[88] A. Ghazy and M. Shalan, “OpenLANE: The open-source digital ASIC implementation flow,” in *Proc. Workshop on open-source EDA technol.(WOSET)*, 2020.

[89] J. Cong and Y. Ding, “FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994, doi: 10.1109/43.273754.

[90] B. W. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970, doi: 10.1002/j.1538-7305.1970.tb01770.x.

[91] C. M. Fiduccia and R. M. Mattheyses, “A linear-time heuristic for improving network partitions,” in *19th design automation conference*, 1982, pp. 175–181. doi: 10.1109/DAC.1982.1585498.

[92] S. K. Lim, *Practical problems in VLSI physical design automation*. Springer Science & Business Media, 2008.

[93] R. Rajaraman and D. F. Wong, “Optimum clustering for delay minimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 12, pp. 1490–1495, 1995, doi: 10.1109/43.476579.

[94] 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. doi: 10.1145/266021.266273.

[95] J. A. Roy, S. N. Adya, D. A. Papa, and I. L. Markov, “Min-cut floorplacement,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1313–1326, 2006.

[96] C. Sechen and A. Sangiovanni-Vincentelli, “The TimberWolf placement and routing package,” *IEEE Journal of Solid-State Circuits*, vol. 20, no. 2, pp. 510–522, 1985, doi: 10.1109/JSSC.1985.1052337.

[97] M.-C. Kim, D.-J. Lee, and I. L. Markov, “SimPL: An effective placement algorithm,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 50–60, 2011.

[98] J. Lu *et al.*, “ePlace: Electrostatics based placement using nesterov’s method,” in *Proceedings of the 51st annual design automation conference*, 2014, pp. 1–6.

[99] P. Spindler, U. Schlichtmann, and F. M. Johannes, “Kraftwerk2—a fast force-directed quadratic placement approach using an accurate net model,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1398–1411, 2008.

[100] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, “RePlAce: Advancing solution quality and routability validation in global placement,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 9, pp. 1717–1730, 2019, doi: 10.1109/TCAD.2018.2859220.

[101] S. Do, M. Woo, and S. Kang, “Fence-region-aware mixed-height standard cell legalization,” in *Proceedings of the 2019 on great lakes symposium on VLSI*, 2019, pp. 259–262. doi: 10.1145/3299874.3318012.

[102] K. Han, A. B. Kahng, and J. Li, “Optimal generalized h-tree topology and buffering for high-performance and low-power clock distribution,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 478–491, 2020.

[103] C. Y. Lee, “An algorithm for path connections and its applications,” *IRE Transactions on Electronic Computers*, vol. EC–10, no. 3, pp. 346–365, 1961, doi: 10.1109/TEC.1961.5219222.

[104] J. Soukup, “Fast maze router,” in *Proceedings of the 15th design automation conference*, 1978, pp. 100–102.

[105] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968, doi: 10.1109/TSSC.1968.300136.

[106] F. O. Hadlock, “A shortest path algorithm for grid graphs,” *Networks*, vol. 7, no. 4, pp. 323–334, 1977, doi: `https://doi.org/10.1002/net.3230070404`.

[107] K. Mikami and K. Tabuchi, “A computer program for optimal routing of printed circuit conductors,” in *IFIP congress*, 1968.

[108] D. Hightower, “A solution to line-routing problems on the continuous plane,” in *DAC ’69*, 1969.

[109] F. K. Hwang, “On steiner minimal trees with rectilinear distance,” *SIAM Journal on Applied Mathematics*, vol. 30, no. 1, pp. 104–114, 1976, Accessed: Nov. 19, 2022. [Online]. Available: `http://www.jstor.org/stable/2100587`

[110] M. Hanan, “On steiner’s problem with rectilinear distance,” *Siam Journal on Applied Mathematics*, vol. 14, pp. 255–265, 1966.

[111] J.-M. Ho, G. Vijayan, and C. K. Wong, “New algorithms for the rectilinear steiner tree problem,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 2, pp. 185–193, 1990, doi: 10.1109/43.46785.

[112] G. Robins and J. S. Salowe, “On the maximum degree of minimum spanning trees,” in *Proceedings of the tenth annual symposium on computational geometry*, 1994, pp. 250–258. doi: 10.1145/177424.177978.

[113] G. Georgakopoulos and C. H. Papadimitriou, “The 1-steiner tree problem,” *Journal of Algorithms*, vol. 8, no. 1, pp. 122–130, 1987.

[114] M. Borah, R. M. Owens, and M. J. Irwin, “An edge-based heuristic for steiner routing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 12, pp. 1563–1568, 1994, doi: 10.1109/43.331412.

[115] W. C. Elmore, “The transient response of damped linear networks with particular regard to wideband amplifiers,” *Journal of applied physics*, vol. 19, no. 1, pp. 55–63, 1948.

[116] K. D. Boese, A. B. Kahng, and G. Robins, “High-performance routing trees with identified critical sinks,” in *Proceedings of the 30th international design automation conference*, 1993, pp. 182–187.

[117] K. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins, “Rectilinear steiner trees with minimum elmore delay,” in *Proceedings of the 31st annual design automation conference*, 1994, pp. 381–386.

[118] L. McMurchie and C. Ebeling, “PathFinder: A negotiation-based performance-driven router for FPGAs,” in *Proceedings of the 1995 ACM third international symposium on field-programmable gate arrays*, 1995, pp. 111–117. doi: 10.1145/201310.201328.

[119] M. Pan, Y. Xu, Y. Zhang, and C. Chu, “FastRoute: An efficient and high-quality global router,” *VLSI Des.*, vol. 2012, Jan. 2012, doi: 10.1155/2012/608362.

[120] A. B. Kahng, L. Wang, and B. Xu, “Tritonroute: The open-source detailed router,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 3, pp. 547–559, 2020.

[121] A. B. Kahng, L. Wang, and B. Xu, “The tao of PAO: Anatomy of a pin access oracle for detailed routing,” in *2020 57th ACM/IEEE design automation conference (DAC)*, 2020, pp. 1–6.

[122] *QFlow 1.3: An Open-Source Digital Synthesis Flow.* R. Timothy Edwards, 2018. Available: `http://opencircuitdesign.com/qflow`

[123] “KLayout 0.28.2 documentation.” 2022. Available: `https://www.klayout.de/doc-qt5/klayout.pdf`

[124] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, “The magic VLSI layout system,” *IEEE Design & Test of Computers*, vol. 2, no. 1, pp. 19–30, 1985.

[125] *LEF/DEF 5.7 Language Reference.* Cadence Design Systems, Inc.., Nov. 2009. Available: `http://www.ispd.cc/contests/18/lefdefref.pdf`

[126] E. J. McCluskey and F. W. Clegg, "Fault equivalence in combinational logic networks," *IEEE Transactions on Computers*, vol. 100, no. 11, pp. 1286–1293, 1971.

[127] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE transactions on Computers*, vol. 30, no. 3, pp. 215–222, 1981.

[128] H. Fujiwara and H. Ozaki, "A fanout oriented test generation algorithm for combinational circuits," *Transactions of the Information Processing Society of Japan*, vol. 24, no. 1, pp. 56–63, 1983.

[129] M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A highly efficient automatic test pattern generation system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 1, pp. 126–137, 1988.

[130] T. Larrabee, "Test pattern generation using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, 1992.

[131] M. Abdelatty, M. Gaber, and M. Shalan, "Fault: Open-source EDA's missing DFT toolchain," *IEEE Design & Test*, vol. 38, no. 2, pp. 45–52, 2021.

[132] H. K. Lee and D. S. Ha, "Atalanta: An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagation," in *1991, proceedings. International test conference*, 1991, p. 946.

[133] T. Spyrou, "Opendb, openroad's database," in *Proc. Workshop on open-source EDA technology (WOSET)*, 2019, p. 6.

[134] "Intel® arria® 10 device overview." Intel Corporation, 2022. Available: `https://www.intel.com/content/www/us/en/docs/programmable/683332/current/adaptive-logic-module.html`

[135] M. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+ nextpnr: An open source framework from verilog to bitstream for commercial fpgas," in *2019 IEEE 27th annual international symposium on field-programmable custom computing machines (FCCM)*, 2019, pp. 1–4.

[136] M. Gort and J. H. Anderson, "Analytical placement for heterogeneous FPGAs," in *22nd international conference on field programmable logic and applications (FPL)*, 2012, pp. 143–150.

[137] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt, "CRoute: A fast high-quality timing-driven connection-based FPGA router," in *2019 IEEE 27th annual international symposium on field-programmable custom computing machines (FCCM)*, 2019, pp. 53–60.

[138] S. Dhar, M. A. Iyer, S. Adya, L. Singhal, N. Rubanov, and D. Z. Pan, "An effective timing-driven detailed placement algorithm for FPGAs," in *Proceedings of the 2017 ACM on international symposium on physical design*, 2017, pp. 151–157.

[139] P. de Vos, "How to fuzz an FPGA - my experience documenting gowin FPGAs." Chaos Computer Club, 2021. Available: `https://www.youtube.com/watch?v=P7o6KcbpjmQ`

[140] 1BitSquared, "Glasgow interface explorer." CrowdSupply, 2020. Available: `https://www.crowdsupply.com/1bitsquared/glasgow`

[141] D. Petrisko *et al.*, "BlackParrot: An agile open-source RISC-v multicore for accelerator SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020.

[142] R. T. Edwards, M. Shalan, and M. Kassem, "Real silicon using open-source EDA," *IEEE Design & Test*, vol. 38, no. 2, pp. 38–44, 2021.

[143] *Alliance/Coriolis VLSI CAD Tools.* LIP6 - Laboratoire d'Informatique de Paris 6. Available: `http://coriolis.lip6.fr/`

[144] G. Gouvine, "Coloquinte: A mixed-size placer for the coriolis toolchain," PhD thesis, LIP6 - Laboratoire d'Informatique de Paris 6, 2015.

[145] G. Gouvine, "Coloquinte: A mixed-size placer for the coriolis toolchain." LIP6 - Laboratoire d'Informatique de Paris 6, 2015. Available: `https://github.com/Coloquinte/Coloquinte_placement`

[146] K. P. Ghosh and A. K. Ghosh, "Technology mediated tutorial on RISC-v CPU core implementation and sign-off using revolutionary EDA management system (EMS)—VSDFLOW," in *2018 china semiconductor technology international conference (CSTIC)*, 2018, pp. 1–3.

[147] A. B. Kahng, M. Kim, S. Kim, and M. Woo, "RosettaStone: Connecting the past, present and future of physical design research," *IEEE Design & Test*, 2022.

[148] E. Wang, "HAMMER: A platform for agile physical design," *Master's thesis, EECS Dept, UC Berkeley*, 2018.

[149] E. J. Tywoniak and J. Harder, "Yosys GitHub pull request 3504: Temporal induction counterexample loop detection." 2022. Available: `https://github.com/YosysHQ/yosys/pull/3504`

[150] E. J. T. Ondrej Ille, "CTU CAN FD IP core issue 416: Formal verification of CTU CAN FD." 2022. Available: `https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core/-/issues/416`

[151] E. J. Tywoniak and T. Gingold, "GHDL issue 2204: LAST_VALUE and friends in PSL cause a crash after throwing an error." 2022. Available: `https://github.com/ghdl/ghdl/issues/2204`

[152] D. Sporn, whitequark, and E. J. Tywoniak, "Amaranth GitHub issue 704: Detect and reject netlists with combinatorial loops." 2022. Available: `https://github.com/amaranth-lang/amaranth/issues/704`

[153] E. J. Tywoniak and whitequark, "Amaranth GitHub pull request 726: Lib.coding: Remove GrayDecoder comb loop for consistency." 2022. Available: https://github.com/amaranth-lang/amaranth/pull/726

[154] E. J. Tywoniak, "Yosys GitHub pull request 3519: smt2/smtbmc: Fix mathsat counterexample VCD dump crash." 2022. Available: https://github.com/YosysHQ/yosys/pull/3519

[155] E. J. Tywoniak and N. Engelhardt, "Yosys GitHub pull request 3546: Add missing memory width assert preventing division by zero." 2022. Available: https://github.com/YosysHQ/yosys/pull/3546

[156] "CTU CAN FD IP core." FEE CTU, 2023. Available: https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core

[157] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *2016 IEEE/ACM international conference on computer-aided design (ICCAD)*, 2016, pp. 1–6.

[158] E. J. Tywoniak, "OpenLane fork." 2023. Available: https://github.com/ekliptik/OpenLane/tree/view-placement

[159] E. J. Tywoniak, "OpenLane and OpenROAD experimental scripts." 2023. Available: https://gitlab.com/tywonemi-school-stuff/physical-design-scripts

[160] T. Gingold, "Ghdl-yosys-plugin: VHDL synthesis (based on GHDL and yosys)." 2023. Available: https://github.com/ghdl/ghdl-yosys-plugin

[161] M. Venn, "TinyTapeout." 2023. Available: https://tinytapeout.com/

[162] E. J. Tywoniak, "TinyTapeout 02 submission: Chase the beat." 2022. Available: https://github.com/ekliptik/tt02-chase-the-beat

[163] E. J. Tywoniak, "OpenLane issue 1517: All random operations should take a seed." 2022. Available: https://github.com/The-OpenROAD-Project/OpenLane/issues/1517

[164] E. J. Tywoniak, "OpenROAD fork." 2023. Available: https://github.com/ekliptik/OpenROAD/tree/hacks

[165] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2007.

[166] "Cadence JasperGold." 2022. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[167] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI physical design: From graph partitioning to timing closure*. Springer Science & Business Media, 2011.

[168] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI test principles and architectures: Design for testability.* Elsevier, 2006.