2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

The Three Body Problem

There's more to building Silicon than what EDA tools currently help with.

Ben Marshall, PQShield
Bristol, UK
(*ben.marshall@pqshield.com*)

Peter Birch, VyperCore
Bristol, UK
(*peter@vypercore.com*)

Keywords—EDA; Tooling; Flows; Standardisation; Open Source;

I. Introduction

Electronic Design Automation (EDA) tooling facilitates the development of cutting edge technologies at the nanometre scale, providing the means to rationalise an incredibly complex task into abstractions simple enough to become second nature to an engineer. In a post-Moore's Law environment, an increasing number of companies are turning to bespoke Application-Specific Integrated Circuits (ASICs) to achieve the same generational performance gains that consumers have come to expect. The extreme costs of a failed tape-out rightly demands high levels of assurance from verification and physical analysis, leading to long development timescales. From our personal experience (not to mention that of our colleagues), companies tend to develop layers of internal tooling and processes around the major EDA vendors' offerings to help reduce these risks. However, this raises a question of if there is a missing piece of the puzzle?

Our intention with this paper is to properly describe this "missing piece" and highlight some of the issues with current EDA tooling that make it difficult for engineering teams to fill it in. We then propose some potential solutions that could improve the situation for users and vendors alike.

To explain our motivation, we ask you to consider the software industry and its distinctly different approach to collaboration and how a more "open" approach is beneficial to everyone involved. Take for example the incredible communities surrounding GCC [1] and LLVM [2], these two compiler suites dominate the industry for C and C++ compilation and yet are entirely free and open source. Apple, Qualcomm, and Google, along with many other well known names, both sponsor and help maintain LLVM [3], and Intel has even shifted their optimised compilers to use the project [4].

Databases are also comparably complex systems for storing data that need to be both performant and extremely fault tolerant as they handle important and sensitive information. Up to the early 90s, the space was dominated by commercial solutions such as Oracle Database [5] (introduced 1979) and Microsoft Sequel Server [6] (1989). Open source alternatives arrived in the mid-90s with MySQL [7] in 1995 and PostgreSQL [8] in 1996, and now these solutions have taken over the market with PostgreSQL and MySQL having a substantial lead over both Microsoft and Oracle according to StackOverflow's 2022 Developer Survey [9].

Our intent with these comparisons is to highlight how open source software is beneficial to engineers, while still providing vendors plentiful opportunities to market tools and support services that build on such ecosystems. In our view, existing EDA tooling tends to provide monolithic toolchains that take a "walled-garden" approach to data management and how they interface with other tools and flows, often relying on proprietary, unstable and undocumented data formats. We are concerned that this approach is failing to keep pace with the challenges presented by modern IP and ASIC development.

A methodology frequently touted by the major EDA vendors is the "shift-left" of late issue discovery [10,11,12] to earlier points in the development cycle, where such issues are generally easier (and cheaper) to fix. In practice implementing this across a design flow requires the engineering process to work coherently, as it cannot be fully realised within a single tool's domain. For example, resolving a clock crossing issue is far easier

while the RTL is under active development than at the point it has reached physical layout - but a robust "shift-left" means that CDC needs to produce a consistent result on both RTL and netlist, with the netlist often being produced by a tool from a different vendor.

Any engineer involved in design flow development will be familiar with the challenges of tool integration and data management - problems for which little standardised tooling or methodology exists. In our experience this has led to companies developing vast webs of bespoke shell, Makefile, Perl, and Python scripts that are jealously guarded, often by teams who just want to get on with the hardware design task and don't have the time to make the most considered decisions. These scripts become critical to the success of each tapeout or delivery, and yet are often been borne out of urgency without deep consideration for their structure or extensibility. It is surprising that, unlike other industry wide problems such as testbench design, there isn't an agreed approach.

In this paper we suggest that the problem is not the tools themselves, but the lack of emphasis on how they are stitched together. We believe it is this ability to create smooth transitions between stages in a wider flow that is sorely lacking at present. That is, we recognise *building hardware is as much a problem of data management as it is of microarchitectural design or verification, and our tools should recognise it too*. We will also describe how the existing interfaces to tools make these junctions so difficult, how we unconsciously deal with these tooling deficiencies (i.e. abstracting them under a mountain of scripting), and what better interfaces should look like. We will provide a set of concrete recommendations to standards groups which might aid this situation, allowing for more flexible and innovative EDA solutions.

We acknowledge that in some areas the open source community has already risen to the challenge, for example Olof Kindgren's award-winning FuseSoC [13] and Edalize [14] abstract away the difficulties of assembling and processing IP (from the Edalize Readme: "[...] all these tools are doing this in completely different ways and there's generally no way to import configurations from one simulator to another. Dread not! Edalize takes care of this for you"). We believe that the industry has much to learn from such tools and concepts.

In this paper we will define three tenets that we believe underpin a robust and capable design flow, and describe how commercial EDA tooling sometimes prevents engineers from achieving these aims as easily as they could. We explore commonly used solutions and why these are unsuitable for modern ASICs, and discuss how solutions from other domains might provide inspiration. The paper goes on to explore how the industry and EDA vendors could help address these issues, and what we propose as a general solution. Finally we discuss Blockwork, an open source design flow that will address many of the points discussed here, and which we hope could act as the "missing piece" in all our tool suites.

II. Expressing Design Flows as a Graph

A design flow is composed of a series of steps that progressively transform RTL into different forms of results, be that lint warnings, functional coverage databases, or even the final GDS-II delivery. The number of such pathways quickly increases as a flow is developed, and the interdependencies quickly become complex.

At a high level a flow can be summarised as being a collection of different sequences, where each sequence is formed of a series of transformations that each take a number of inputs and produce a number of outputs. Certain transformations may be common between different sequences, while others may be unique. One way to express such an assembly is an acyclic graph [15] where transformations are represented by the nodes of the graph and inputs and outputs by the edges.

Figure 1 shows a simple design flow terminating in lint, coverage merge, and synthesis - in reality there would be many further transformations, but this suffices as an example. There are six formats of file represented by edges in the graph (`.sv`, `.sv.mako`, `.py`, `.tcl`, `.exe`, and `.db`), mostly originating from the filesystem but some are generated by the templating, compilation, and simulation transformations. There are seven types of transformation, including reading from the filesystem as represented by an identity transform [16].

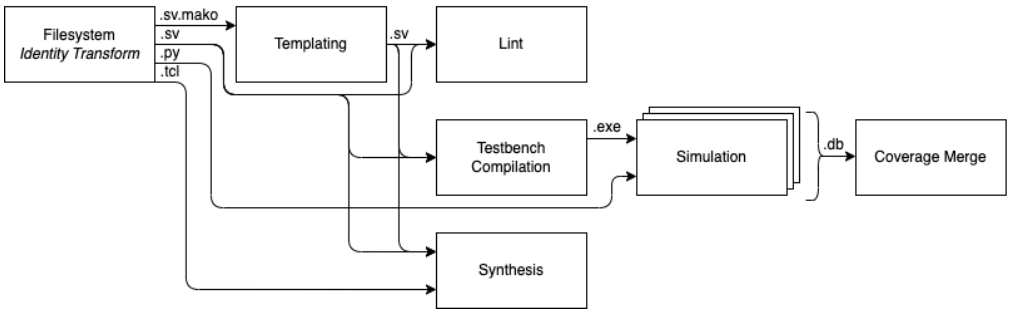

Figure 1: A simple design flow expressed as a graph

This example shows only the minimal steps in a silicon design flow, and yet the complexity of passing information between tools is already apparent. As a design flow is developed the depth, complexity, and interconnectedness of the graph only ever increases. If incorrectly structured, this quickly becomes difficult to manage and dependencies of any particular step become hard to discern. If there is no confidence in dependency tracking, the only way to be sure of a result is to re-run the entire graph, which quickly becomes expensive.

III. Three Tenets Of Silicon Design

To try and frame the complexity of this problem more concretely, we introduce our three tenets of silicon design: traceability, modularity and reproducibility. In this section we will discuss each of these terms and show how current EDA tooling fails to meet these goals, and how effort is currently spent on filling this gap.

Traceability

Traceability allows us to know the provenance of any particular engineering artefact at any stage of the silicon design process. Without this, we don't know what we're building, whether we delivered the right version of the RTL to the physical design team, or sent the right GDS file to be manufactured. Being able to trace the complete set of input and intermediate artefacts used to produce any given output (which itself will likely be an input to another step) is essential to our notion of traceability. We call this *referential* traceability.

Because chip design is a process spread across time, it is also essential to know the version of inputs fed into each transform. This requires some way of tagging artefacts (inputs or outputs from transforms) with information from the version control system (Perforce, Git, SVN) being used. In a simple project, this level of traceability might be realised simply by checking out the project code at a single version and running the entire implementation flow. In a larger project, with multiple deliveries of sub-components feeding into a larger system, possibly across different version control systems or even companies, tracing the version of a component quickly becomes more complicated. We call the tracing of artefact versions across time *temporal* traceability.

Being confident in the provenance of a given artefact requires trusting how operations are traced - maintaining a link to the underlying version control system for as long as possible provides a very strong reference, that we believe is essential. Other less robust schemes, such as encoding a version number in a filename or text file, provide opportunities for accidental or even malicious substitutions to be made, the latter being critical when securing a supply chain. Building such a capability into a build system means that traceability is inherent, rather than relying on process and discipline.

In our experience, current EDA tooling (somewhat understandably) does not engage with these requirements. Vendors generally focus on addressing individual steps of the silicon design process (simulation, synthesis, linting, DFT insertion, etc) rather than the more abstract problem of tracing and managing the artefacts between these stages. It is also a much more difficult tool to build (or, product to sell), because it will necessarily be opinionated about how each transformation in the design process is composed and the form of artefacts passed between them. As a result, companies build their own ad hoc management systems, and suffer

the burden of their maintenance. Engineers moving from one company to another must become familiar with yet another bespoke set of tools that achieve exactly the same aim as those at their previous company, but in a different way. The industry should recognise this situation as similar to how UVM introduced a standardised approach to functional verification.

Modularity

Modularity refers to being able to compose transform steps easily and correctly. Recalling our model of silicon design as a series of transformations on build artefacts, we want a way to express this chain of transformations, and ensure that each intermediate artefact in the chain of transformations is of the right type.

This is similar to what a standard Makefile does. Indeed, we often see engineers trying to solve this problem with Makefiles, before realising that it doesn't quite provide what's needed and resorting to Python or Perl instead. For example, imagine a high level process for collecting code coverage from a block using a UVM testbench as shown in figure 2 - there are several vendor agnostic steps involved:

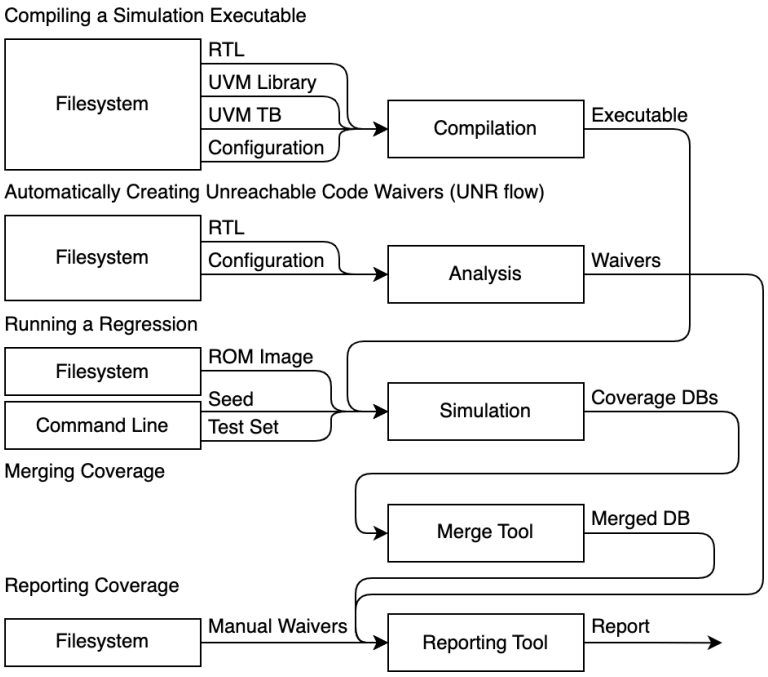

Figure 2: A simple flow for code coverage collection

This set of steps should be familiar to any verification engineer, or designer who's suffered the manual waiver creation process. Every EDA vendor has a particular implementation of this process with tools to support each particular step, and the overarching flow is implemented in a different way at every company.

Additionally, because each step often produces outputs in a format shared by no other tool (possibly with no other tool from the same vendor), considerable effort is often spent writing scripts to parse, process and transform these tool specific outputs into something which can be used for the next transformation in the process. For example, *there is no good reason* that coverage waivers cannot be vendor agnostic. The information needed to create a coverage waiver relates to the design and is not tool or vendor specific, yet every tool (whether commercial or open source) has a unique way of specifying them. Companies that want a single source of truth for coverage, lint, or other waiver must spend effort creating a vendor agnostic representation and then transforming that into the vendor specific representation. Is it too much to ask to have one agreed way to do this? We think not.

Reproducibility

Reproducibility ensures that a design flow run by engineer A on Monday on machine X will work identically for engineer B on Friday on machine Y. In other words, every process should be deterministic, producing the same output for the same inputs with all of those inputs under the explicit control of the engineer.

The problem does not just stop with explicit inputs to a particular process, script or EDA tool. Often, reproducibility extends to the entire operating environment for a design flow, including the Operating System, standard libraries, environment variables and in some cases even specific hardware to run on. This leads to very conservative development environments, where changes to any of the tool versions requires extensive testing and coordination. It's also essential that the same environment can be exactly recreated months or even years later when debugging or re-creating an issue seen in a manufactured device. EDA vendors currently specify the environment in text description and rely on the user to recreate it, meaning that companies must bear the cost of qualifying the environment each time they install or update a tool.

IV. Why Does Current Tooling Fall Short?

The inherent risk involved in silicon tapeouts means that the industry is generally conservative in adopting the latest tools and trends and, given the costs involved, this is understandable. Vendors are aware that customers can be extremely reluctant to adopt new or significantly updated products, meaning that tool interfaces generally change in slow and predictable ways to give users time to adapt.

The major vendors market products that address mostly the same market segments, with subtle differences in capabilities or performance. In general, what is possible with one vendor is possible with another. It would, therefore, seem possible for standardised command line and scripting interfaces to exist. However, vendors may view their interfaces as superior to the competition, and a key selling point to their customers. In truth the differences are often sources of confusion and frustration, and provide a barrier to adopting a better solution. Searching for the "magic flag" which makes the entire flow work is not an experience anyone enjoys.

Non-Disclosure Agreements (NDAs) between vendors and their customers also provide a significant hurdle to any collaboration or knowledge sharing between companies, as the terms often prohibit any sharing of command line flags and switches, with documentation hidden within paywalled support sites. This means that companies must obscure the most useful details of their flows when sharing them publicly, meaning others must discover the same pitfalls for themselves. One only needs to consider the utility of StackOverflow to understand the importance of community surrounding a product, and the current NDAs prevent this from blossoming.

As projects become increasingly complicated, tool interfaces that are predominantly designed for human interaction and not for automation quickly become problematic. Inputs, outputs, and control are provided in proprietary formats that differ significantly not only between vendors but also between products from the same vendor. Such formats are generally not documented, and manual or programmatic editing outside of the vendor's environment is strongly discouraged. While this approach may not seem problematic in isolation, in the context of a long and complex design flow it does little to assist the end user. A tool which helps you "shift left" is no good if you have to spend considerable effort transforming inputs and outputs with custom scripts.

V. Current Solutions

The lack of standardised interfaces between tools and general difficulty in constructing complex and robust design flows leads to each company to, at great length and cost, build its own bespoke system that is then as jealously guarded as the crown jewels. In many cases, even if the organisation was so inclined to publicly share their flow, the level at which vendor tools pervade the infrastructure would often make it very difficult to expose without violating an NDA. That said, there are a few examples where individuals or organisations have successfully shared infrastructure - a few such examples are:

- cocotb [17] is a Python framework for writing testbenches that aims to work with every commercial and open source RTL simulator, with the flow transparently adapting to different simulators;
- SiliconCompiler [18] focuses on automating synthesis and physical layout of designs, predominantly targeting open source tools (e.g. Yosys) but can be extended to support commercial tools;
- Olof Kindgren's FuseSoC [13] is a package manager for building hardware from predefined components while another of Kindgren's tools, Edalize [14], provides an abstraction layer for interfacing with various free or opensource tools such as Icarus Verilog [19] and Xilinx Vivado [20]. We note that the sales pitch for Edalize is that it will "save you from having to deal with *the boring stuff of interfacing the EDA tools yourself*" (emphasis ours).

While fantastic, each tool only addresses a limited portion of the design, verification, and implementation process. Discontinuities in workflows are often the source of misunderstandings and frustration, so these tools will then need to be wrapped in a wider build system such as GNU's Make [21] or Google's Bazel [22]. However, from experience the depth and complexity of transformations that need to be applied to a hardware design are ill-suited to such software-focussed build systems.

For example, an SoC will contain multiple bespoke components each of which needs to be linted, DFT inserted, simulated, synthesized, equivalence checked, and more. Many of these tasks then need to be re-run when moving up the hierarchy to the top of the chip, taking different sections and configurations of the design. This quickly leads to an explosion in the number of pathways and artefacts being managed.

An interesting comparison to make is with data engineering tools such as Apache's Airflow [23] or Elementl's Dagster [24] that orchestrate deep and complex pipelines of transformations, routing artefacts between steps. There are well documented APIs for stages and pipelines and standardised formats for artefacts. However, such tools are intended for processing continually streamed analytics data, for example spotting trends in customer purchases. These tools expect data pipelines to run continuously on a server, rather than on demand.

There are important lessons to take away from both build systems and data engineering tools, and for EDA workflows a hybrid of the two that takes the interactive and on-demand nature of a build system with the ease of pipelining of a data engineering tool could be incredibly powerful.

VI. What Should We Aim For?

Every transform must be traceable, reproducible, and modular, and the relationship between inputs and outputs must be well understood. All artefacts should be serialisable and ideally vendor agnostic, although some files (for example compiled simulation binaries) will remain inherently vendor specific.

Similarly, the interfaces to invoke tools should conform to agreed standards. For example many tools will read in a netlist with search paths and defines, where subtle differences between tools frequently catch users out. Tools operating in the same domain (e.g. simulation, formal proof, synthesis) could conform to a standard interface for specifying constraints, libraries to load, and so forth. With such changes, a user could be confident that infrastructure developed for one vendor would port to another with minimal effort.

However, these changes are not subtle and would require vendors to agree on common standards. Even if this is possible, it is unlikely to be a rapid decision with a long tail of transition. Therefore this leads us to the question of what we can do today that moves us closer to these goals?

Few, if any, existing build systems address the complexity of a modern ASIC and so something new is required. A perfectly capable flow can be built for a single IP using GNU Make, but scaling up to an ASIC constructed from internal and third party components often leads to custom layers being progressively added. An ideal tool would make it trivial to manage complex, reusable, and interconnected pipelines of transformations. This is not a trivial effort and the result could be of benefit to the whole industry. Sharing such a tool would allow for community adoption and contributions, rather than each company reinventing the wheel.

Tools should be encapsulated such that they run in their 'ideal' environment and, as far as possible, interface with preceding and successive transformations using standard, non-vendor specific formats. Inputs and outputs from each tool invocation should be strongly defined, so that dependencies can be traced throughout the build graph. The flow should be explicitly aware of supporting tools and libraries from the environment, and those that have not been specified should be obscured from the transformation (i.e. if a tool only requests Perl, then it may not call Python). This will achieve the properties of traceability and reproducibility, providing that the tool itself is deterministic given constant inputs. Modularity would require 'specialisation' and 'normalisation' layers before and after the tool is invoked to transform to and from vendor-specific formats.

EDA vendors should support the industry by allowing integrations of their tools with build systems to be shared publicly - this would require the relaxation of NDAs. This would lower frustration for the customer and allow them to become productive faster, while the vendor's support burden could be reduced as there will be fewer questions on how to get a tool running. In an ideal world, vendors would maintain the tool integrations into the build orchestrator themselves so that "best practice" is guaranteed. Vendors should also make sure that their inputs and outputs are easily machine readable, with the formats well documented.

Industry engagement is essential to convince vendors that this will be beneficial for both them and their customers. We believe this should be an open discussion, with companies contributing their feedback, criticism, and time to develop these shared standards. We are an industry upon which modern infrastructure and livelihoods are built. We solve some of the most difficult engineering challenges on earth. Standardised data formats and consistent command line arguments should not be too much for us as an industry to deliver.

VII. A Concrete Implementation: Blockwork

Blockwork is a free and open source build system built to tackle the complexity of designing, verifying, and implementing a modern ASIC comprising many different components. While extremely flexible, the tool also offers a recommended methodology on structuring a hardware repository and its flow - for example:

- Hardware development throughout design, verification, and physical implementation should be managed in a single monolithic repository - numerous articles [25,26] document the benefits.
- The OS, supporting software libraries, and tool versions must be under the flow's control so that the build "environment" is always well understood.
- Flow orchestration should be implemented in few languages with configuration separated from execution. In Blockwork's case configuration is specified in YAML and execution is coded in Python.

Blockwork isolates the build environment from the host machine using containerisation. It is built around the Docker API [27] and is compatible with container runtimes including Docker and Podman. The "foundation" container image is based on Rocky Linux 9, with minimal packages installed into the OS itself. Tool binaries are selectively "bound-in" from the host based on the activity being performed, for example a transformation may request only GCC and GNU Make. The tool may also modify the shell environment, for example extending **`$PATH`**.

Isolation is also used to selectively bind context-specific files and folders from the host system - for example if a project has two independent IPs, **`acme_cpu`** and **`acme_dma`**, then when an engineer simulates **`acme_cpu`** the files for **`acme_dma`** are completely inaccessible. This operates at a fine granularity, meaning that different activities on the same top-level can only access files that they request in advance. This leads to strict enforcement of dependencies, thus increasing confidence in the traceability and reproducibility of any given transformation. When transformations are chained together, outputs from one stage can be automatically bound-in as inputs to the next.

Containerisation is used to 'normalise' the file system. For example, sections of the repository will be bound (read-only) to paths under **`/project`** based on activity, while tools are bound under **`/tools`**, and work is

performed under **`/scratch`**. This isolates activities from the exact configuration of the host, allowing an engineer's laptop and a server farm to appear identical aside from compute performance. As a side benefit, this means if a third party tool encodes an absolute path into an output, consecutive steps can then be run on anywhere by anyone without updates.

A **`bw`** command offers a common entrypoint for launching builds, running tools, or even 'bootstrapping' the workspace. Detailed documentation [28] explains each command and what they are doing under the hood, for example the following command opens a wave viewer to explore a VCD:

```
$> bw tool gtkwave.view ../hardware.scratch/acme_cpu/sim/waves.vcd
```

Blockwork is under rapid development with the current focus on defining a syntax for constructing the complex pipelines for lint, verification, synthesis, and ultimately delivery of a design to GDS-II. The project is open source on GitHub (github.com/blockwork-eda/blockwork) and welcomes feedback and contributions.

VIII. Conclusions

Our hope is that we inspire vendors and engineers to consider the end-to-end requirements of complex design flows, and how the tools we use day-to-day could be improved to simplify the task of integration. Our main desires are that interfaces are standardised, file formats are vendor agnostic and well documented, and the relationship between inputs and outputs of any tool are strongly defined. We strongly believe that relaxing NDAs between vendors and customers to an extent that allows tool integrations to be shared would be beneficial to all parties involved, and is the foundation for establishing common practices and tooling for design flows. We offer Blockwork as a stepping stone on this road. We believe that it is built on a solid foundation and will develop over time into a robust and capable build system for even the most complex hardware projects.

References

1. Free Software Foundation, Inc., "The GNU Compiler Collection", gcc.gnu.org last accessed 23/04/2023
2. LLVM Foundation, "The LLVM Compiler Infrastructure Project", llvm.org last accessed 23/04/2023
3. foundation.llvm.org/docs/sponsors
4. www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html
5. Oracle, "Database", www.oracle.com/uk/database last accessed 23/04/2023
6. Microsoft, "Microsoft Data Platform", www.microsoft.com/en-us/sql-server last accessed 23/04/2023
7. MySQL, "MySQL", mysql.com last accessed 20/08/2023
8. PostgreSQL Global Development Group, "PostgreSQL: The World's Most Advanced Open Source Relational Database", www.postgresql.org last accessed 23/04/2023
9. survey.stackoverflow.co/2022#most-popular-technologies-database-prof
10. blogs.sw.siemens.com/xcelerator/2023/01/20/what-is-shift-left/
11. www.cadence.com/en_US/home/company/cadence-academic-network/educators/shift-left-methodology.html
12. www.synopsys.com/automotive/what-is-triple-shift-left.html
13. O. Kindgren, "FuseSoC User Guide", fusesoc.readthedocs.io/en/stable/user/index.html last accessed 23/04/2023
14. O. Kindgren, "Edalize User Guide", edalize.readthedocs.io/en/latest/user/index.html last accessed 23/04/2023
15. www.britannica.com/topic/graph-theory
16. Cuemath, "Identity Function", www.cuemath.com/algebra/identity-function last accessed 21/08/23
17. cocotb, "Python Verification Framework", www.cocotb.org last accessed 21/08/23
18. SiliconCompiler, "Compile code into silicon", www.siliconcompiler.com last accessed 21/08/23
19. S. Icarus, "Icarus Verilog", github.com/steveicarus/iverilog last accessed 21/08/23
20. AMD Xilinx, "Vivado ML Overview", www.xilinx.com/products/design-tools/vivado.html last accessed 21/08/23
21. Free Software Foundation, Inc., "GNU Make", www.gnu.org/software/make last accessed 21/08/23
22. Google, "Bazel", bazel.build last accessed 21/08/23
23. The Apache Software Foundation, "Apache Airflow", airflow.apache.org last accessed 21/08/23
24. Elementl, Inc., "Dagster", dagster.io last accessed 21/08/23
25. J. Hanlon, "Silicon infrastructure", www.jameswhanlon.com/silicon-infrastructure.html last accessed 21/08/23
26. D. Luu, "Advantages of monorepos", danluu.com/monorepo last accessed 21/08/23
27. Docker, Inc., "Develop with Docker Engine API", docs.docker.com/engine/api last accessed 21/08/23
28. Blockwork EDA, "Blockwork build environment", blockwork.intuity.io last accessed 21/08/23