HAMMER: A Platform For Agile Physical Design

Edward Wang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-28
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-28.html

May 1, 2020

Acknowledgement

This work was funded in part by the DARPA CRAFT program (HR0011-16-C-0052).

Contents

1 Keywords

VLSI, software engineering (SE), physical design, digital design, CAD/EDA, design methodology

2 Motivation and Background

2.1 Background

After looking at the ENIAC, the state of the art computer in 1945, it would be difficult to imagine that this monstrous beast could ever fit inside the tip of a pen. ENIAC contained approximately 18000 vacuum tubes (electronic switches), weighed 30 tons, occupied a 30x50 foot room, and consumed about 150 kW of power (equivalent to operating about 37 electric ovens at the same time[1]). At its peak operating efficiency, the ENIAC could perform about 5000 instructions (calculations) per second.[2][3]

In contrast, the Intel Core i7-8700K, a representative top-end desktop computer CPU in 2017, is capable of executing about 39,000 million instructions per second using 2 billion[1] transistors (electronic switches), and consumes only 95W of power[5]. The total chip package weighs 3 ounces, and its dimensions are less than 4x5 inches[2].[6]

This miracle of computing is usually explained via the phenomenon known as Moore's law. In the 1960s, Gordon Moore (co-founder of Intel) made an observation (now known as Moore's law) that the number of transistors on a computer chip doubles about every two years. His observation has held true until recently, bringing computing from the crude machines of the 1960s/1970s to today's smartphones, intelligent thermostats, and data centers.[7]

To first order, Moore's law was significant in that it predicted performance would double, while at the same time, area and power consumption would remain constant. With this power of doubling, software developers were able to add more features and produce reasonably-performant software in a shorter amount of time. This phenomenom occurred because they could now program in increasingly productive higher-level languages (e.g. Java instead of assembly/machine code), focus on non-performance aspects of computing such as user interfaces, and exert less man-power on low-level performance optimization.[8] As Jeff Atwood noted in 2008, "until the day that Moore's Law completely gives out on us, one thing's for sure: hardware is cheap - and programmers are expensive."[9]

Recently, however, Moore's law has slowed in delivering the promised performance gains. A companion trend called Dennard scaling allowed transistors to shrink in size and consume proportionally less power, meaning that the number of transistors per area could quadruple, while the chip as a whole still consumed the same amount of power.[10]. Around the mid-2000s, however, Dennard scaling was the first trend to violate expectations when computer chips hit the single-core "power wall" - the phenomenon where a chip would consume more power than could be adequately dissipated via a heatsink. This meant that clock speeds and single-threaded performance could no longer follow the exponential gain described by

[1]The actual number is unclear since Intel does not publish official transistor counts for this chip - estimates range from 1 to 3 billion transistors.[4]

[2]Retail dimensions include consumer packaging and other overhead.

Moore's law. Since then, processor vendors like Intel have continued to pack more transistors onto individual dies by increasing the number of cores instead of trying to increase the clock frequency for a single core.[11]

Unfortunately, even Moore's Law itself is under siege. As transistors approach atomic limits where classical physics break down, we can observe the difficulties in process costs - e.g. photolithography, tooling, and yield all become challenging, resulting in higher fabrication costs. Despite the shrinkage of transistors, the cost per million gates stops dropping after 28nm. In fact, the cost per million gates actually starts to slightly increase.[12] [13] Recently, the semiconductor community has been debating whether there will be any significant process nodes past 7nm, heralding the death of Moore's law.[14] While many exotic (non-CMOS silicon-based) technologies such as carbon nanotubes and quantum computing exist, they do not function in the same environmental conditions with the same performance and accuracy characteristics.Hence, since software can no longer automatically rely on Moore's law (either via single or multi-core scaling), we must develop custom hardware and accelerators to continue reaping performance gains.

Chip making, however, is very expensive, ranging from $50-$300 million for conventional chip projects[14]. While manufacturing each chip costs some money, this amount may not account for most of the cost in low to moderate-volume products, which is dominated by non-recurring engineering costs (NREs) - one-time overhead costs required to design a chip which remain constant regardless of the number of units manufactured. In particular, for large chips, the cost is dominated by design complexity in all areas (software, hardware design, and physical design). This design complexity results in high engineering effort which translates to high costs. For example, the CEO of a fabless semiconductor company noted that "engineering salaries are by far the biggest cost in SOC design". [15]

Developing software for programming new semiconductor devices dominates the cost of new products, followed by hardware costs like verification, validation, IP qualification, and physical design. Fabrication and other overhead (e.g. marketing) come afterwards.[16] [15] In other aspects of new semiconductor projects such as software, re-use helps bring down cost, while current physical design methodology prevents physical design efforts from being effectively re-used across nodes/projects/tools. Other work such as domain-specific languages like Halide [17] can help reduce the cost of developing software for new semiconductor products. This work focuses on addressing the hardware-related portions of the NRE.

On the hardware front, we hypothesize that a number of factors contribute to high hardware design NREs, such as sub-optimal programming interfaces in CAD tools (TCL), inefficient abstractions, and CAD tool vendor oligopolies. Another factor which could contribute to high hardware design NREs is the disconnect between RTL and physical design. By creating a platform for agile hardware design, we hope to tackle the hardware design NREs. This work focuses mainly physical design, but also addresses validation, IP qualification, and verification concerns by exploring inefficient programming interfaces and poor abstractions.

Why do we need these additional abstractions? RTL alone is not sufficient to tape out a chip. RTL provides details regarding logic that goes on the chip, but almost completely abstracts the physical implementation of the chip.[3] The information required for physical

[3]Some CAD tool vendors support custom Verilog inline attributes to specify certain constraints, but as we will discuss later, this is not portable nor necessarily desirable.

implementation includes placement, clocks, power, port constraints, pads/bumps, but also off-chip/integration concerns like packaging. While automatic place and route tools can do most of the logic cell placement, large hard macros such as memories or analog blocks need to be manually placed[4] to achieve optimal efficiency. To tape out any real chip, this information needs to be passed on to the CAD tools somehow.

While there are some existing solutions, they still prevent hardware innovation by making the NRE cost high, mostly by non-portable or inefficient methodologies. Even in the presence of previous tapeouts, creating a new chip project remains inefficient and frustrating, which translates to high design engineering costs. We want to examine why this inefficiency occurs, and propose a modular[5] platform to enable hardware innovation and make new chip projects less frustrating.

2.2 Existing Solutions Are Inadequate

Many large institutions have created internal scripts and flows[6] to create chips. These scripts hard-code information from 3 different sources that are mixed in together: 1) Technology process design kit (PDK) (Process Design Kit) paths and references; 2) RTL/project-specific paths; 3) CAD tool vendor-specific commands. In addition, many scripts are commonly implemented using programming languages like GNU Make, bash scripting, sed, and TCL. They lack abstractions like object-oriented programming, type systems, well-defined data structures, well-defined libraries, etc, which, at least from anecdotal evidence, makes maintaining them very difficult or frustrating[7].[8]

While CAD tool vendors provide "example flows" or "reference methodologies" intended to be somewhat modular, the fact that customizations and additions to the flow happen directly in the flow by modifying or writing TCL commands/variables limits re-use by tying those commands to a specific CAD tool vendor, technology node, and chip project.

Some vendors have custom inline attributes in Verilog which encode physical annotations to CAD tools (e.g. FPGA tools, VCS, etc). These attributes are non-standard[9], and variable

[4]Some CAD tool vendors may have some automatic macro placement but for the most part higher-level design information is required to do this realistically.

[5]By modular here we mean that there should be a well-defined API to make it easy and safe to add new plugins for supporting technologies and CAD tools without requiring a comprehensive understanding of all the environment/shell/Make variables floating in a global namespace. We will discuss this further in the Existing Solutions and design sections.

[6]Most of these scripts are non-general, institution-specific, and even if they were well engineered, they are not freely available, hindering re-use.

[7]While frustration and errors are hard to quantify, some commit messages (written by multiple individuals) from an internal chip repository include "Fixing [person]'s stupid typos", "fixing carriage return in [person]s block", "Derpa doo", "CAPS LOCK CUZ I HAVE ANGER ISSUES AND IM STILL SALTY THAT FLIPCHIP IN [CAD tool] IS THE WORST", "Jesus f*** [CAD tool]", "Last edit. Last commit. Plzzz" (of course, there were (many) more fixes required afterwards), and "Stupid stupid stupid".

[8]PLSI was a previous attempt at creating a modular flow. During our attempt to use it for a tapeout and maintaining it afterwards, we found it difficult to avoid making errors like simple typos, global variable dependencies, esoteric syntax, etc that made it hard to use. In addition, the flow was very monolithic and it was difficult to re-use portions of it but not others.

[9]Section 4.4: "There are no standard attributes in the Verilog-2001 standard; Software tools or other standards will define attributes as needed."[18]

support across vendors makes it non-portable. Furthermore, not all physical design traits are possible to specify in inline extensions (e.g. no major ASIC tool vendor supports encoding clocks or SRAM placement).[10] Therefore, encoding physical design information in inline attributes is not a fully portable solution, and as we will see later, even if we could modify the source RTL to include attributes/annotations, this may not necessarily increase re-use.

Other previous attempts tended to follow the framework pattern as opposed to the library pattern. The key distinction between frameworks and libraries is the idea of "inversion of control"[19] - in the library pattern, the API provides a collection of functions/methods we can call, and the control flow is defined in our application. In contrast, the "framework [pattern] embodies some abstract design" pattern and programmers use it through a series of bindings or hook points in which to add application-specific functionality, and control is passed to the framework via some form of "main loop".[20] Fowler (2005) calls this phenomenon the Hollywood Principle - "Don't call us, we'll call you".[20][11]

In terms of a VLSI flow, a framework-style approach involves calling the flow's "main function" (or entry point), and the flow is customized by injecting project-specific logic into certain binding/hook points which the flow will call at the appropriate point.Unfortunately, in the context of a VLSI flow, a framework approach makes it difficult to gradually adopt the tool, since the framework makes some assumptions about the abstract design of the flow that may not always hold when we are actually trying to use the tool. Here are a few examples of situations where a framework-style design is problematic:

- Writing a design space exploration loop for a sub-step of place-and-route (e.g. clock tree synthesis) - this will be difficult/annoying to do with a framework which assumes a linear run through synthesis and place-and-route without repeating any steps in between and does not provide programmatic, easy-to-use APIs to control the steps. In contrast, a library would allow us to write such a program very easily since we can control when portions of tools run.

- Integrating a build system for managing CAD tool runs - integrating complex build systems which are often framework-style proves difficult when both the VLSI tool and the build system assume ownership of control.

- Customizing specific methods and processes within the flow - the flow may embody a specific model for adapting generic RTL to technology-specific RTL suitable for tapeout (more on this later), but the project may have its own specific ways of generating or adapting the RTL to the technology, and having a tool which assumes ownership of calling those processes will be hard to re-use.

- Generating small digital blocks for analog - our VLSI flow may assume an abstract model of generating a full digital-top chip with top-level I/Os, but we may want to use the tool to only generate some small analog blocks without any top-level integration or checks, resulting in a conflict of what the flow model should be.

[10] To our knowledge.

[11] Some examples of framework-style libraries include GUI libraries and JUnit.[20]

When a framework lacks sufficient binding/hook points for fully controlling its behaviour, often the only recourse is to create a fork (a local copy of the code) and modify those portions of the code. This causes a large software maintenance burden since the modifications need to be kept up to date with the upstream code in order to benefit from bug fixes and new features in the upstream code.[21]

Nonetheless, we recognize that frameworks exist and are useful because patterns do in fact repeat[19], and that not every use of a library is completely different. In order to address these needs, we build libraries that have both modular (library-like) and high-level (framework-like) APIs. However, the high-level APIs should be as thin as possible. The high-level APIs make it easier to get started with using the tool by embodying common patterns, and the modular APIs make it easy to build custom functions and extensions using the library. If we only provided modular APIs but no high-level APIs, there would be a proliferation of copy-paste in integrating all these modular APIs, making it difficult to add new features. For example, if customizing a place-and-route flow required creating a fully custom function and running each step individually, it would be difficult to re-use upstream changes e.g. when a new step is introduced.

Another challenge faced in creating VLSI flows is being backwards-compatible with the underlying languages/technologies we are trying to abstract - in this case, TCL and Verilog. Many programming languages owe some to all of their success in being backwards-compatibility with existing languages. For example:

- Chisel features compatibility with Verilog (the pre-existing technology) in both ways - it allows users to use Verilog modules with a BlackBox mechanism[22], and while Chisel's modules elaborate to Verilog by default, it provides mechanisms for users to define modules with fully customized port names and without the implicit clock and reset ports.

- Scala is Java-compatible to first-order, making it possible to use existing Java libraries/frameworks and develop libraries/frameworks which can be used in Java.[23][24]

- C (with many compilers) is backwards-compatible with assembly language, making it possible to (relatively) easily write and call assembly language routines from C.[25]

- High-level programming languages like Julia and Python all have C extensions. Languages not able to re-use optimized GEMM libraries will be at a disadvantage. Even Java has JNI to call native code and take advantage of optimized libraries in 'legacy' languages.

Hence, we aim to design a system which allows the use of TCL/Verilog for backwards-compatibility and adoption, while providing a platform to make it possible to encode as much of physical design knowledge at a higher level. Similarly, we aim to create a physical design IR that can either be generated from agile tools like FIRRTL/Chisel or be hand-written. In doing so, we aim to provide the infrastructure to use higher-order (e.g. Scala-based) generators for physical design information without forcing their use.

Finally, we aim to create a platform which is modular. With existing compiler frameworks like LLVM, instead of writing $M * N$ pieces of code for M input languages and N output

formats, we can write $M + N$ pieces of code instead: M front-ends for input languages, and N code generation backends. Likewise, through our hardware IR, we aim to enable many different hardware "input passes" to generate physical design constraints that can be consumed by a number of different backend tool and technology plugins. For example, instead of writing a specific program to generate clock constraint TCL fragments for a specific CAD tool vendor from rocket-chip, we can leverage FIRRTL and our hardware IR to generate generic constraints which can be portably consumed by many different plugins.

2.3 Agile Hardware Design to the Rescue

Electrical engineering/software engineering inherited waterfall-style planning from older engineering disciplines such as mechanical and civil engineering, where once execution of a project was started, it would be very difficult and expensive to change course.[26] In contrast, a group of software engineers published the Agile Manifesto in 2001, valuing small, close iterations, adaptiveness to change, and producing working software.[27] The software industry has benefitted from agile development - more specifically, from agile's recommendation to use small teams/projects and break down large projects into smaller projects - a 2013 study referenced by Fox and Patterson indicates that small projects are cancelled 4% of the time, in contrast to large projects which are cancelled 38% of the time.[28]

A traditional ASIC design process (e.g. as exemplified by [29] or [30]) involves a few sequential stages performed in order, waterfall-style. First, requirements are gathered for the chip project, and an architectural design is created at a block diagram level along with specifications for each of the major blocks. This is typically called the architectural stage of design. Next, this high-level architectural block diagram is passed to a group of engineers who implement blocks in the architecture at the RTL level (or schematics for analog designs) and then run simulations to verify that the blocks are correct[12]. Finally, the RTL and schematics are turned into a layout either using place-and-route tools for digital, or via manual layout for analog.[13] This is followed by post-layout simulations, final signoff (DRC/LVS), and then post-silicon test and bring-up.

We aim to bring agile ideas such as minimum viable product, fast feedback loop, and adaptable tools and systems to chip design. As we have shown in the Background section, conventional chip-making endeavours are $50-300 million projects, well above the $1 million threshold used by [28]. However, software is malleable - in principle, it can in principle be updated following initial deployment. In contrast, it is much harder to just "update" a custom chip broadly[14] deployed in datacenters, in the field, etc.[15] Nonetheless, the fabrication phase is still relatively short (say approximately 3-6 months) and most of the NRE cost and inefficiencies occur during the design phase of the project. As we note later, even producing a fabricatable chip design without actually fabricating it (called a tapein) is still

[12]Exactly how "correct" a design needs to be and how we can show/prove a design is correct has a degree of freedom/controversy.

[13]The spoiler alert here is that analog design is becoming more agile via generators like BAG[31].

[14]This also implicitly assumes the "old Moore's law" business model of few products and high volume, but the same difficulties could also apply to limited-quantity, custom silicon like the TPU.

[15]Of course, microchips can be programmable and have firmware/microcode, but that comes with its own set of deployment/security/logistical challenges.

very valuable for providing physical design/implementation feedback to other parts of the chip building endeavour. Chip design is still very much waterfall-based, which makes it hard to be adaptable, and increases the cost of designing a chip when changes need to be made.

Agility, defined as the ability for a project to quickly adapt to change, is essential both from a business perspective and a technical perspective. As a 2011 Harvard Business Review article notes, companies that "thrive are quick to read and act on signals of change". In particular, they note that companies which can experiment, manage complex systems, and mobilize quickly have an advantage from adaptability in the marketplace.[32][16] These traits are also embodied in agile ideas - for example, adaptable tools and systems can help manage complexity, and the fast iteration loop allows us to quickly experiment and provide preliminary results for faster mobilization.

On the technical side, making a chip is a complex set of negotiations between all aspects of chip design, including process technology parameters, physical design, analog/mixed-signal blocks, and IP blocks/macros. For example, during a system-on-chip (SoC) design, the amount of space allocated to each portion of the chip varies over time as design progresses. But existing design mentality (e.g. freezing RTL) prevents changes and makes SoC design difficult, especially when later stages in the "waterfall" require changes to the chip. Instead of being agile and adaptive, the idea of fixed/frozen designs ("design freezes") being "thrown over the wall" to another team ties the hands of downstream teams and prevents their feedback from being effective in helping upstream teams in designing for downstream considerations.

For example, physical design (traditionally thought of as a "downstream" step from architecture/RTL) problems often require architectural changes to fix properly:

- During the BROOM (BOOMout-of-order processor with memory resilience) tapeout, a specialized register file was unimplementable with any reasonable QoR (space and critical path wise), requiring an architectural re-design to split the register file and break up the critical paths. Likewise, a cross-bar in the design turned out to be unroutable and required conversion to a tri-state cross-bar to be implementable.[34]

- During the Hurricane (codename for Berkeley DVFS research demonstration chips) tapeouts, not getting early feedback on how big certain cross-bars would be or how much congestion it would create resulted in delays in taping out and lowered QoR.[17]

- During the SPLASH (ASIC spectrometer) tapeout[35], we encountered the so-called SPLASH "ring of death"; a large ROM was mapped to gates in a way that does not work well for the place-and-route tool.

Generator-based design can help us achieve our goal of building systems adaptively with a fast feedback loop by facilitating re-use between blocks and projects. By providing a library of pre-written, parametrizable generators, designers can achieve a minimum viable product in

[16] With regards to projections/forecasts, Warren Buffett opines that he has "no use whatsoever for projections or forecasts. They create an illusion of apparent precision. The more meticulous they are, the more concerned you should be."[33] Since the traditional waterfall model involves performing each step without returning to previous steps, it inevitably relies on projections or forecasts about downstream steps, casting doubt on the viability of a purely waterfall model of development.

[17] Unfortunately, the author is not aware of pre-existing publications on this chip at the time of writing.

less time and obtain feedback in order to facilitate the agile feedback cycle. These generators encode a recipe or process for designing a family of circuits (digital or analog) and allow us to re-use work in a way that works for our current project, thereby avoiding the time-consuming phenomenon of having to re-write or modify instances to fit current project needs.[36]

Existing work such Chisel[22] and BAG[31] enable agile hardware design in RTL and analog circuits respectively. However, in order to make a chip, we need a backend VLSI flow which takes RTL, adds technology-specific mappings, and calls CAD tools for synthesis and place-and-route, all while providing the appropriate physical design information along the way. However, as outlined in the Existing Work section, we don't yet have an agile, re-usable, and principled[18] backend flow implementation. As a result, we can't directly leverage the fixes and lessons from previous tapeouts, meaning we have to repeat history by writing the same kind of RTL changes, the same kind of placement, etc over and over again, wasting time - the same problem analog design faced before BAG or digital design before Chisel (re-creating or tediously modifying existing instances from previous work to meet current project needs.)

More specifically, we recognize that physical design is a collection of many difficult problems - there is no single silver bullet. Hence, we aim to lower the barrier to solving these problems. Other tapeouts have dealt with these problems, but their solutions are not general or reusable. This makes it more difficult to produce a minimum viable product quickly, causing low adaptivity and a long feedback loop - physical design becomes a bottleneck in the agile hardware design ecosystem in the presence of agile generators like Chisel and BAG. It also explains why, for example, analog designers don't use place-and-route tools for generating useful pieces of logic like decoders, adders, and state machines.[19].

To remedy this, we aim to provide a system to encourage designers to encode solutions in a more reusable way, so future tapeouts can leverage previous work (even with different technologies, CAD tools, or designs). For example, we fixed the "ring of death" and BOOM cross-bar problems with one-shot solutions, but had we encoded those solutions as generators, future designs could leverage those generators/solutions when the same type of issues appeared, reducing the time to minimum viable product.

In order to enable agile physical design, we provide many systems which designers leverage to encode information modularly and build re-usable flows. These include:

- A physical design IR (higher-level and CAD-tool independent directives) to facilitate the generation/description of physical design constraints separately from their underlying expression in imperative TCL commands. This tightens the feedback loop (by not requiring knowledge of tool TCL dialects to write constraints) and increasing adaptivity (by being independent from specific tool expressions).

- An abstraction on top of CAD tools in order to increase adaptivity and manage complexity by separating CAD tool concerns from physical design concerns.

- Higher-level technology abstractions to manage complexity (abstracting away the details of the technology and PDK) and tighten the loop (reducing the amount of time hunting

[18] In terms of modularity, type safety, expressivity, etc. For example, a principled generator can provide huge productivity savings compared to an ad-hoc, one-time, non-reusable solution (Rocket vs OpenPiton).[36]

[19] Anecdotally from speaking to analog designers at Berkeley.

for PDK paths in order to use a technology).

- Pre-provided drivers to capture common patterns of running physical design / VLSI flows and reduce the feedback loop by providing useful, out-of-the-box functions/APIs/shell commands to make getting started easier and faster.
- A mechanism to directly manipulate/introspect on RTL to synchronize physical design constraints with RTL generators and tighten the loop by connecting RTL generators to physical design generators.

We will elaborate further on these in the Design section of this work.

2.4 Challenges

There are a number of possible challenges to this approach.

- Time required to fab is quite large, so automating/optimizing the flow doesn't matter.
  - Tapeouts aren't necessary for evaluation - even doing a "tape-in" (performing the entire design process to the point of generating a fabricatable GDS but without sending it to the fab) will give us good design feedback to improve the architecture and RTL in an agile manner (tight feedback loop, minimum viable product).
  - In academia, it is possible to publish papers with realistic LVS-clean designs.
- Run-times are long on complex designs, so automating/optimizing the flow doesn't matter.
  - It's better to have safe and agile flows that will either work or fail early, rather than manually writing/modifying flows that may or may not work, or relying on copies of scripts of unknown origin.
  - With re-usable generators, we can move towards getting things right the first time or earlier on with checks.
  - We can also iterate on smaller chunks of the design - being agile (e.g. Chisel with Hammer generator), we can easily generate a smaller version of the design that is faster to implement in order to get feedback faster.
  - Also, this is not true for all use cases - for example, when generating digital blocks for analog designers, the bottleneck is the configuration/environment setup and ease of specifying physical design constraints, not the actual place-and-route run time.
- Our abstraction may not always be relevant to everyone.
  - We structure our tool/system in the library-style, enabling users to create custom tools and APIs which build new abstractions relevant to their specific use cases.
  - In addition, we feature explicit backwards-compatibility with TCL, so users can also inject or use custom TCL blobs to achieve specific outcomes for certain cases.

  - Since we aim to build an open source solution, users can always download and modify it (e.g. by making a pull request) to improve the solution. The next time someone faces the same problem, they don't have to solve or encode that aspect of expertise again. Having an open source platform reduces the barrier to sharing and re-using physical design solutions within an organization or in public, just as how the fact that FIRRTL hardware compiler framework[36] and LLVM reduce the effort required to share a compiler pass via standardization.

- If CAD tools can't do a good job without all these extra physical design constraints, then we should fix/optimize the CAD tools to do a good job automatically.[20]
  - Placement[37] and routing[38] are both known to be NP-complete, requiring exponential time. In practice, CAD tools can use good heuristics to achieve decent results quickly enough, but often these optimizations need to be guided with human designer knowledge/insight.
  - Hence, until $P = NP$ is proven, if ever, we still face the problem of passing physical design constraints to CAD tools, and the associated problems of re-usable VLSI flows to pass these constraints. We avoid placing belief in a magical tool/algorithm/compiler that will solve all problems by itself.
  - Even if we could parallelize place-and-route across many machines (assuming it is a fully parallelizable task, which is unlikely), it would still require an exponentially increasing number of machines to perform flat place-and-route (versus hierarchical place-and-route).
  - Furthermore, even if the underlying heuristics were sensible, a hard to compose, hard to use API still creates a barrier to encoding physical design knowledge in a reusable manner. Instead, our platform allows users to build generators an designs using it that are useful for everyone.

- Large institutions may not care as much since 1) they can hire a physical design person who is comfortable with inefficient design practices (their philosophy being either: "not my job and I don't care" or "I got used to the frustrations; it's a part of the job") or 2) have in-house custom/non-reusable scripts that work well enough for their environment.
  - Every institution (whether a company or university) is paying different people to do the same/similar things. If every company/university is going to spend valuable engineer/graduate student time solving the same problem, why not capture this knowledge to save effort on everyone's part? In fact, co-operation/collaboration of this sort has precedence in computing history - in the 1950s, users of IBM 704 machines founded SHARE (Society to Help Avoid Redundant Effort) in order to 1) share programs for the IBM 704 to reduce redundant effort **across organizations**; 2) share knowledge of the workings of the machine to avoid making the the same mistakes across different organizations.[39]

[20] Personally the author would like to see a good open-source, modular, and extensible place-and-route platform that makes it easy to test out new heuristics, just as how LLVM is an extensible platform that allows us to test out compiler passes without having to re-write an entire compiler framework.

  - Engineer hours spent on things that could be fixed with better tools 1) cost $$$ to the organization; 2) waste time - time is money; 3) cause frustration for engineers which translates to stress, lower quality of life, boredom, poor morale, etc.
  - Furthermore, in the agile spirit, we can also enable other members of the team (e.g. architects, RTL engineers, etc) to get feedback earlier and more quickly on their architectures/RTL. For example, computer architects designing caches, accelerators, etc can use our tool to quickly check to first order the viability of their designs. Our tool broadens the accessibility of physical design and VLSI to beyond those who perform physical design for a living.

- Writing TCL/low-level scripts directly don't seem to be that bad. Why spend all this effort developing abstractions?

  - Firstly, there are productivity gains to be had from writing in higher-level languages. Generators yield immense productivity gains, as we've been with BAG[31] generators and the rocket-chipgenerator.
  - More importantly, abstraction is fundamental to enabling us to keep track of and understand the entire work in our mind. Writing software (or software that produces hardware) does not scale well since it means that we must keep track of an increasing amount of information. By using abstraction, we lower the amount of detail we must keep track of in our mind, reducing mistakes and allowing us to focus on the larger picture. As Dijkstra once mentioned - "as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than try to ignore them, for the latter vain effort will be punished by failure"[40].

- Backends are proprietary and non-open source which will hinder re-use anyway.

  - We aim to make it as easy as possible to re-use the core framework between different plugins/site configurations. BAG[31] deals with the same problem - the library itself is open source but it depends on proprietary local configurations and technologies.
  - More importantly, our modular approach allows us to switch between different tool vendors - we will never acheive the same degree of portability by depending only on one tool vendor's APIs. Portability is a valued feature in both academia and industry, and can be achieved by building a better open-source API.
  - Nonetheless, a modular approach is capable of supporting open source backends (e.g. yosys), so it does not detract from potential adoption of open source backend tools.

3 HAMMER: A Platform to Enable Agile Physical Design

3.1 Design Philosophy

HAMMER stands for Highly Agile Masks Made Easily from RTL, and here is our design philosophy.

- Lego Blocks: Instead of a monolithic framework or tool, we provide a series of abstractions, tools, and DSLs which work together to enable agile physical design. Modularity is king. Each part of the system should be able to be used independently from the rest, to the maximum extent possible, in order to increase adoption and usability.
- Incremental Progress: Evolution, not revolution. Provide ways for designers to progressively adopt the system without requiring an all-or-nothing. Provide suitable escape hatches to allow fine-grained custom control where it matters, or where abstractions are incomplete. We believe it is better to have to hardcode 30% than 90%, or to be 50% agile rather than 0% agile.
- Safe, Principled Foundations: Use modern programming languages' benefits (DSLs, type systems, object-oriented programming, software libraries, etc) to reduce errors in the design specification and implementation.

3.2 Overview

An agile physical design flow using the full Hammer methodology starts with design entry in Chisel.[22] Next, we run Chisel code to generate a circuit in the FIRRTL IR. We then run the FIRRTL compiler including technology-mapping/physical design generation transforms and user-written physical design generators (e.g. a floorplan generator), in addition to the standard FIRRTL to Verilog transforms. This generates a technology-mapped Verilog file ready for synthesis and place-and-route as well as fragment(s) of physical design information in Hammer IR. Next, the technology-mapped Verilog, physical design information, driver control information, and environment/tool settings are passed to the Hammer Driver (via the Hammer CLI Driver) which uses the Hammer Tool Abstraction library (hammer-vlsi) as well as the Technology Abstraction library to load and configure the required CAD tool plugins. Finally, depending on the type of action (e.g. synthesis, or synthesis and place-and-route, etc), the appropriate tool plugin is called to generate and run TCL scripts to implement its action subject to the given physical design constraints and tool settings.

Having said that, as outlined in our Philosophy section, we believe in incremental adoption. Instead of forcing users to use the entire Berkeley Scala-based (Chisel/FIRRTL) ecosystem, we allow users to start using backend Hammer abstractions (Tool, Technology, and Driver) with manual design entry in Verilog and manually written physical design constraints in YAML/JSON (Hammer IR) and incrementally move generation of constraints and designs to being generated from the FIRRTL-based passes and generators. We also support hybrid modes of operation - e.g. generating Verilog from Chisel/FIRRTL and manually writing constraints or using FIRRTL generating some constraints using transforms and writing

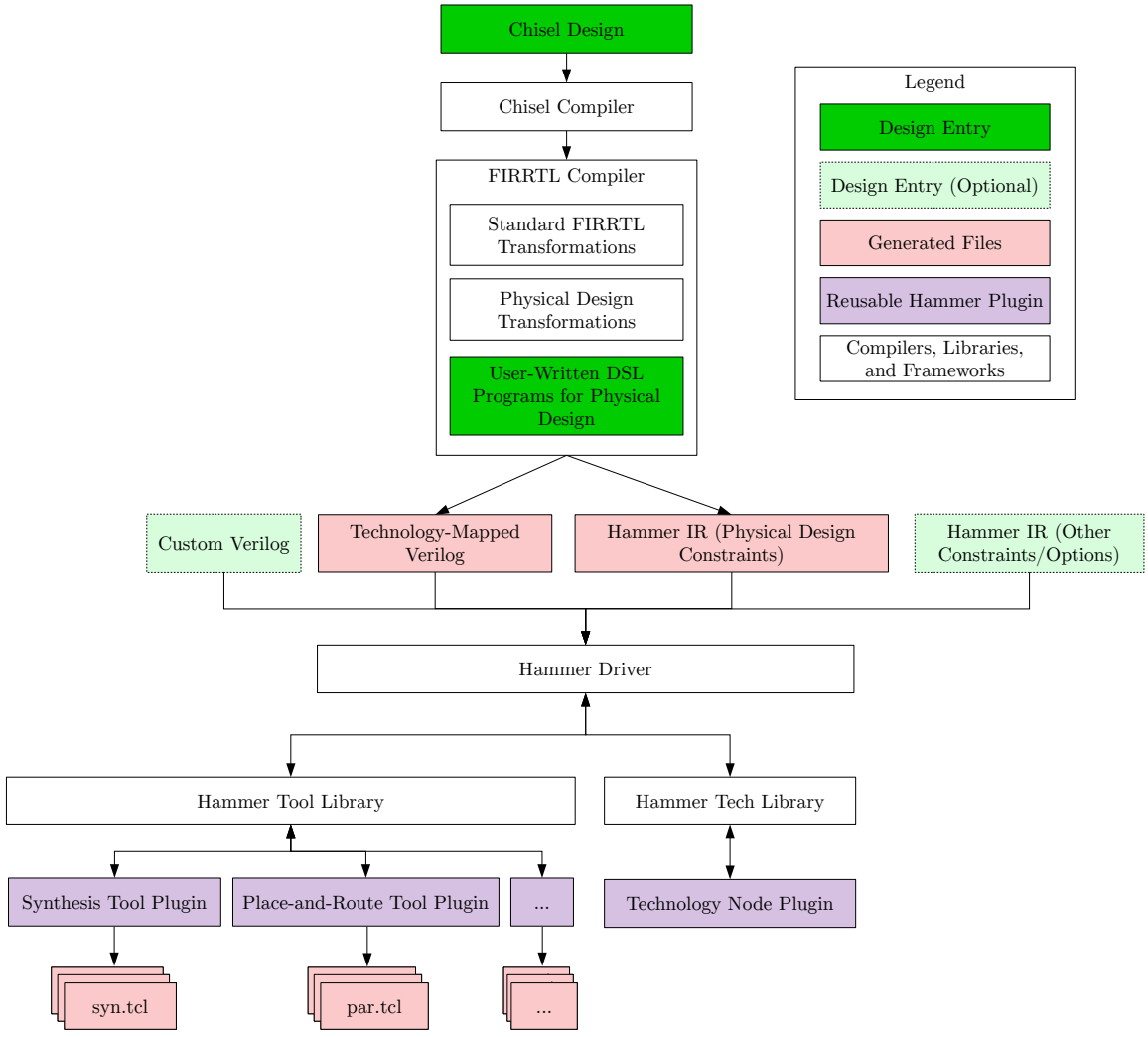

Figure 1: The overall architecture of the Hammer methodology showing the points of design entry, compilers/libraries, generated files/formats, and Hammer plugins, as well as the interactions between them.

other constraints by hand. This allows us to effectively support non-generator (Verilog and manually written physical design constraints) flows, Chisel/FIRRTL-based flows, other FIRRTL-compatible HCLs like Magma[41], or entirely separate modes of design entry (e.g. HLS).

3.3 Python Driver

The Hammer Driver is a Python API which is at a 'higher-level' than the previous abstractions we have seen so far, since we recognized earlier the necessity of providing higher-level/framework-style APIs in order to capture common patterns of abstractions and make it easier for users to get started with our system. The Hammer Driver uses the Driver Control Information and the Tool Control Information (see the IR section for more details) in order to load and instantiate the appropriate plugins in order to run a specific action.

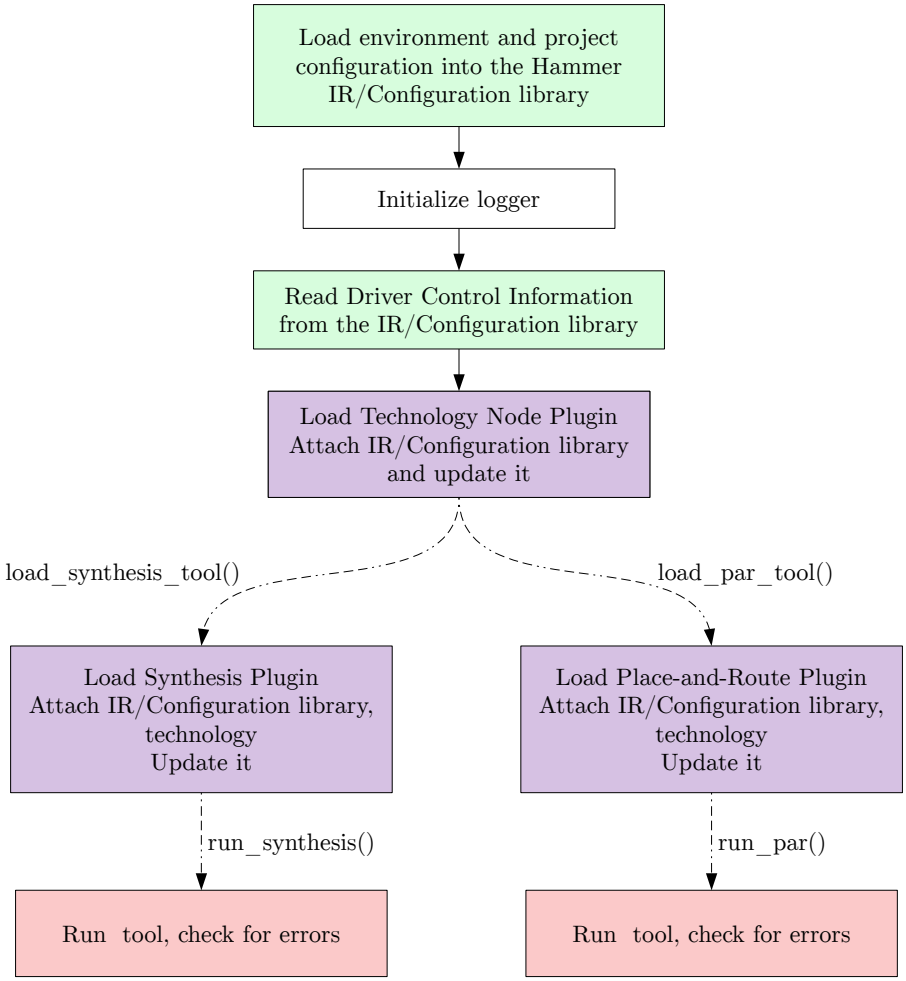

Figure 2: The Hammer driver workflow at a glance.

The Hammer IR/Configuration library recognizes a sequence of IR snippets and combines them together into a large database. The Hammer Driver takes this abstraction a step up and provides a set of well-defined IR snippets and assembles them together in the following chain of precedence to the configuration system:

- builtins: Built-in variables defined by the Hammer Driver.
- core: Core settings defined by hammer-vlsi by default and available for overriding.
- tools: Settings provided by various tools.
- technology: Settings provided by various technologies.
- environment: User-supplied environment settings. (e.g. CAD tool paths, licence servers, etc)
- project: Settings specific to a run of hammer-vlsi.

The Hammer Driver API allows the latter two (environment and project) to be overridden; all other settings are defined by Hammer Tool libraries or the hammer-vlsi core (e.g. builtins, core). After running the appropriate action, the API returns an output copy of the project JSON along with the outputs in order to allow for modular re-use of `hammer-vlsi`. For example, the CLIDriver uses this API in order to implement a function that combines synthesis and place-and-route by using the output settings from synthesis to set the inputs for the place-and-route run.

3.4 Shell/CLI Driver

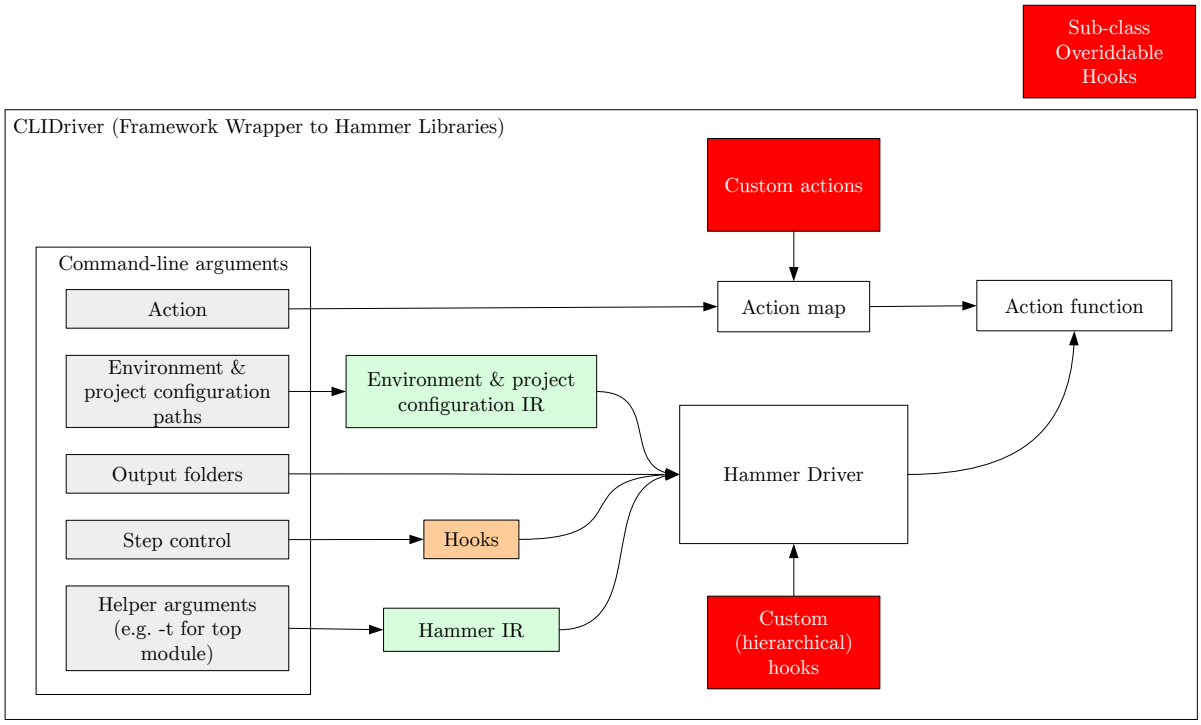

Figure 3: The Hammer CLIDriver framework class is intended to make developing CLI-based flow based on the Hammer Python driver easy.

The command line driver to Hammer, `CLIDriver`, is a framework-style command line wrapper to the (Python) Hammer Driver and the rest of the Hammer Tool Abstractions. The `CLIDriver` has a main function and is callable from the command line. It implements a set of actions which use the Hammer Driver API in order to achieve a specific purpose (e.g. run synthesis, run place-and-route, or run synthesis and place-and-route in the same action. The CLIDriver parses the shell arguments passed to it, sets up the environment and project configurations and passes them to the Hammer Driver for initializing the environment. It also sets the output folders for the Hammer Driver, in addition to parsing step control arguments which get translated into custom hooks and passed to the driver. Finally, helper arguments (e.g. input Verilog files, top module) which are translated to Tool Control Information IR snippets and passed to the Driver.

We created this class in order to 1) make it easy for new users to quickly start using Hammer; and 2) make it easy to add custom hooks and logic to customize the flow. While we recognize the power of the library-style APIs of Hammer in order to build fully custom generators and applications in novel contexts, we want to make it easy to start modifying the flow as a user, and providing this framework is the fastest way to inherit most of the default flow abstraction while slowly adding custom actions, hooks, and features.

With regards to hierarchical support - the default Hammer Tool Abstraction supports a hierarchical mode option but can only handle a single run in order to have maximum separation of concern and lightweightness. Likewise, the Hammer Driver abstraction does not add any additional capability to generate a hierarchical flow, though it facilitates manually writing a hierarchical flow programmatically. The `CLIDriver` interface, however, is capable of automatically generating hierarchical flows from Driver Control Information in the project Hammer IR.

3.5 Physical Design IR / Configuration System

The Hammer IR is the primary data structure used for communicating information to and within the backend of the Hammer methodology. It encompasses four types of information:

- Physical design information - physical design constraints (placement constraints, clock constraints, etc).
- Driver control information - pointers for the Hammer Driver to load the appropriate plugins.
- Tool control information - generic directives for the Hammer Driver on how to load-/launch/configure the tool type (e.g. for synthesis, we have top module, input files, etc).
- Plugin-specific information - settings specific to a given tool/vendor (e.g. licensing, environment variables, tool versions, etc).

The Hammer IR is expressed as a key-value store, consisting of hierarchical keys stored as strings (e.g. "vlsi.inputs.placement_constraints") with scalar (strings, numbers, booleans) or array/list values. Dictionaries cannot be stored directly as values, but can be stored inside arrays used as values. Each key-value pair is referred to as a **setting**.

Keys are strings with a hierarchy (e.g. "vlsi.inputs.test"), separated by periods. Legal key sub-parts are alphanumeric characters and underscores; periods are forbidden since the keys use periods to delimit sub-parts of the overall key. The Hammer IR/Configuration library will "unpack" hierarchies for ease of configuration. For example, the below snippet of IR, when loaded into the IR/Configuration library, will in the following settings (key-value pairs): "vlsi.inputs.supplies.VDD" → "0.9V" and "vlsi.inputs.test" → "bench".

```
{
  "vlsi.inputs": {
    "supplies": {
```

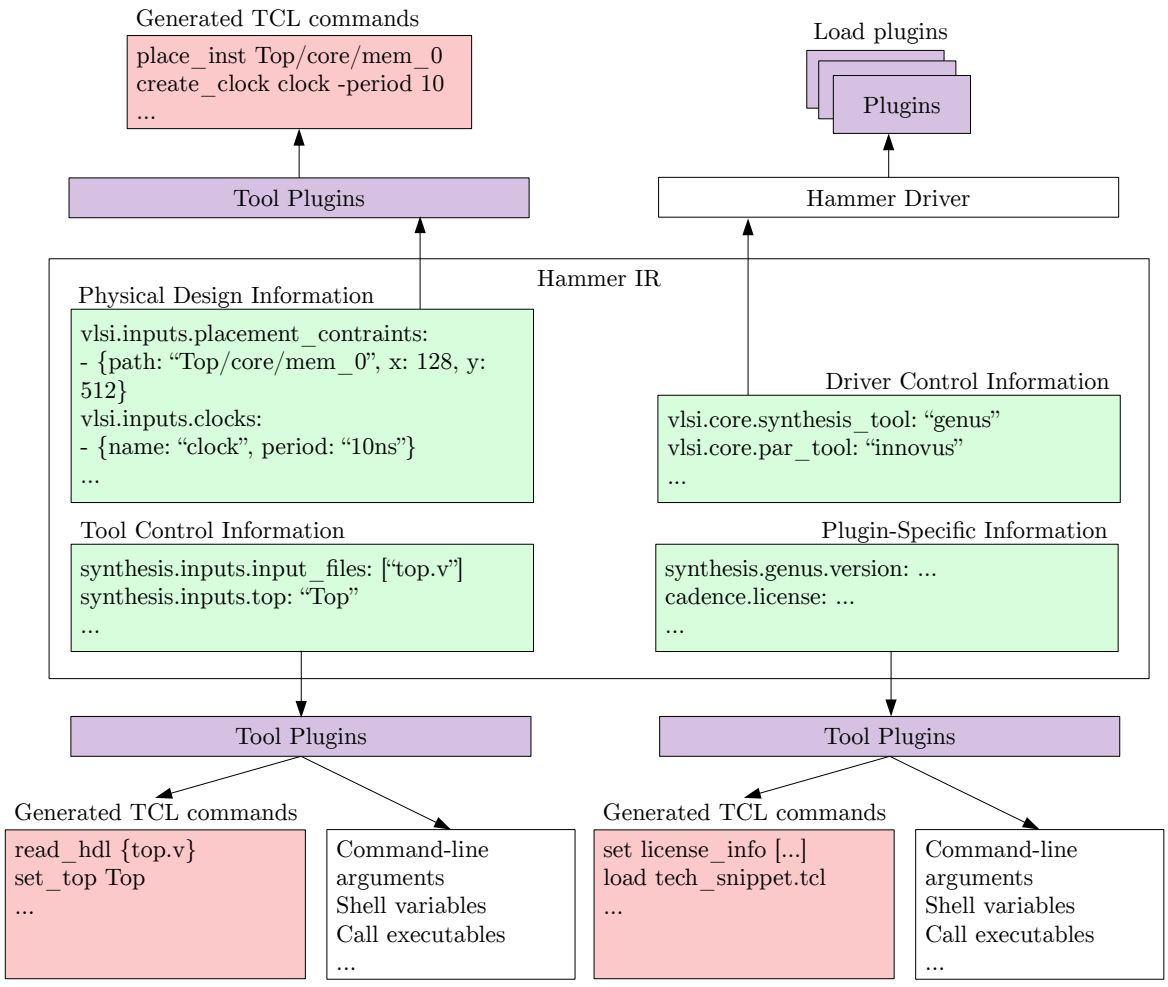

Figure 4: An overview of the four types of information represented in the Hammer IR, stored as a key-value store.

```
      "VDD": "0.9V"
    },
    "test": "bench"
  }
}
```

The IR/Configuration library accepts IR snippets stored in either the JSON (to facilitate programmatic generation of constraints) or the YAML format (to make it easier for humans to write). Configurations in YAML format will be converted to JSON and processed identically. When loading plugins, a default settings file in YAML `defaults.yml` if it exists, will be applied before the JSON default settings file (`defaults.json`).

Note: settings with names like "yes" and "no" do not work since JSON automatically converts them to True/False booleans as keys.

The IR/Configuration library, written in type-safe Python[21]reads in a sequence of IR snippets (YAML or JSON) in order to generate the database, a programmatic key-value store used by other parts of the Hammer methodology to query for information.

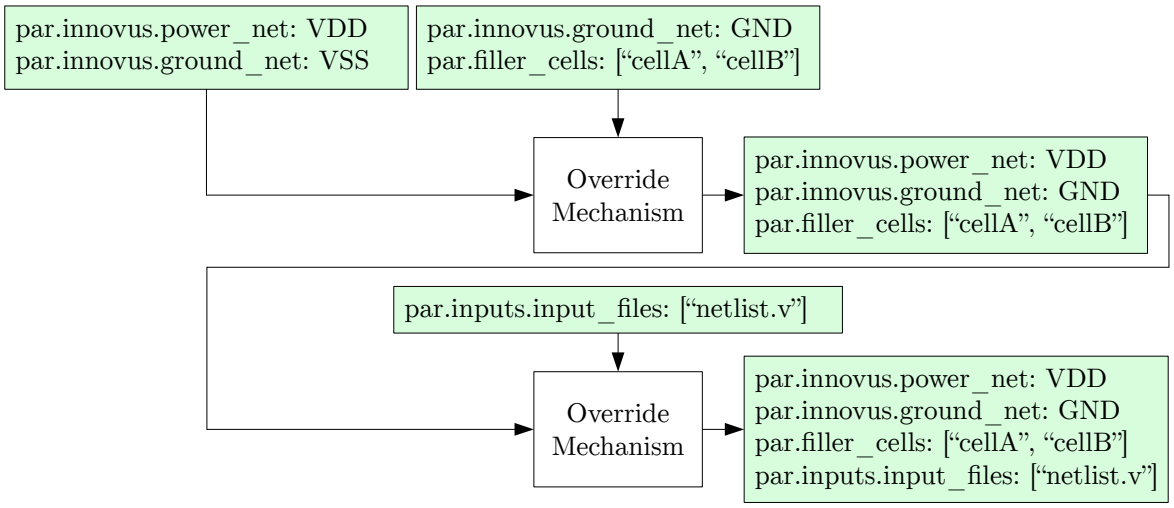

Figure 5: In the IR/Configuration library, IR snippets can override one another.

In order to facilitate re-use, IR snippets which occur later in the sequence of IR snippets will overwrite[22] key-value pairs if also defined in earlier snippets in the sequence. For example, a technology plugin can define parameters with defaults which can be overridden later on in a project IR snippet.

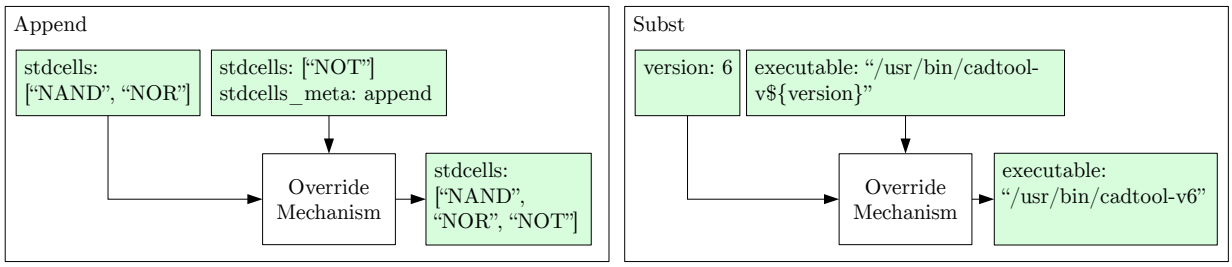

Figure 6: Two examples of meta directives in the IR/Configuration library. Meta directives allow IR snippets to use and modify information from previous snippets.

Meta directives allow IR snippets to use and modify information from previous snippets e.g. by appending to a pre-existing setting or using the value of another setting to create a new setting. When reading a setting, if the setting has an accompanying `_meta` setting, it will be used when getting the setting. The `_meta` setting's value can be either a string of one directive, or an array of multiple directives, processed in order.

The currently valid meta directives are:

[21]Via the mypy static type checker.

[22]While the implementation at the time of writing does not distinguish between intentional and unintentional overwriting

- `append` - append the elements in the setting to the pre-existing setting.
- `subst` - substitute any variable references (in the style of `${other_setting_key}`) that appear in this setting with other variables. Note that referencing other settings that do not exist is an error.
- `transclude` - replace the value of this setting with the content of the file at the path specified in this setting.
- `json2list` - replace the value of this setting with a list parsed from a JSON representation stored in this setting.
- `prependlocal` - prepend the local path of the config dict to this setting.

While user-defined meta directives are not currently supported by the IR/Configuration library as of the time of writing, they may be supported in a future release of Hammer.

Lazy meta directives allow IR settings to forward-reference or re-use information in 'future' snippets, just as how meta directives allow IR settings to re-use information from previous snippets. Each 'regular' meta directive comes with a corresponding meta directive with the prefix `lazy` appended before its name. For example, `subst` has a corresponding lazy meta directive named `lazysubst`. When the IR/Configuration library processes a lazy meta directive, instead of executing the meta directive's action now, it will treat the setting and its meta setting as regular settings and then process them after all IR snippets in the sequence have been processed.

Lazy meta directives are especially useful when defining plugin defaults that are dependent on user settings which will only be available later in the substitution sequence. In particular, `lazysubst` is useful since the setting/template will not be substituted until all known configs have been bound.

Note that all lazy meta directives are single-stage - lazy meta directives cannot depend on other lazy meta directives.[23] For example, the following (YAML) IR snippet is not supported.

```
message: "${work} is fun"
message_meta: lazysubst
work: "taping out ${what}"
work_meta: lazysubst
what: "chips"
```

3.6 Tool Abstractions

`hammer-vlsi`, the Hammer tool library, consists of a tree of abstract interfaces that provide APIs to perform various backend VLSI functions, including synthesis, place-and-route, DRC, LVS, voltage drop analysis, and timing analysis. `hammer-vlsi` uses the IR/Configuration library, as well as the Technology Abstraction library and is written in type-safe (via mypy) Python to be as lightweight as possible in order to enable users to write powerful plugins for Hammer in a modern programming language.

[23] It may be supported in a future release of Hammer.

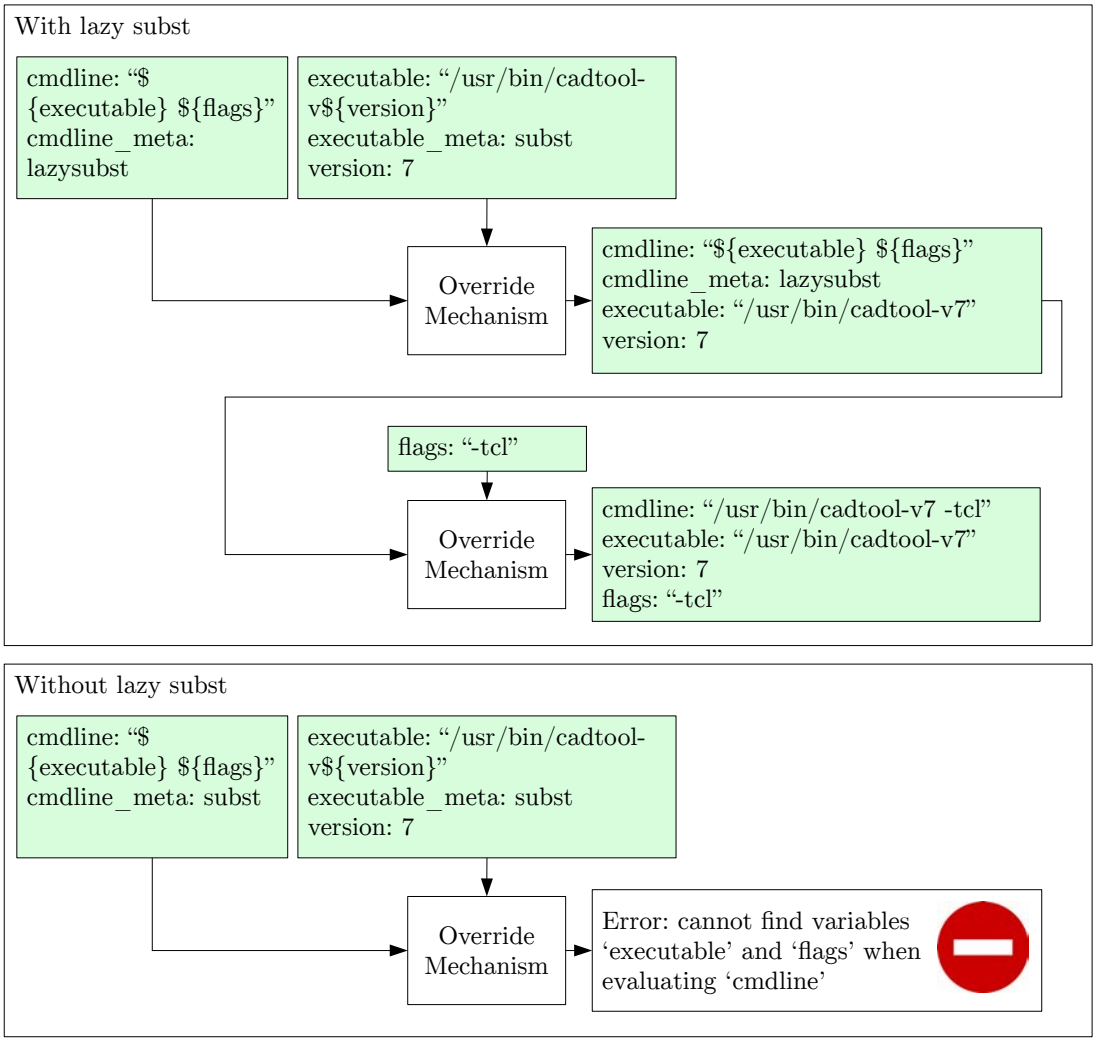

Figure 7: Lazy meta directives allow IR snippets to forward-reference or re-use information in 'future' snippets.

The core abstraction, `HammerTool`, provides a set of common infrastructural methods/APIs useful for writing CAD tool plugins. `HammerTool` contains functions for logging, querying the database of Hammer IR, stage control (see below), accessing the technology, and various helper functions including querying for PDK libraries, string manipulation, and shell functions.

`HammerTool` abstraction structures each CAD tool's flow as a series of steps. This makes it easier to have a fast iteration cycle by making it possible to selectively run steps in the tool without always having to run the entire step. In addition, in order to provide backwards compatibility with TCL, we developed a mechanism (referred to as TCL hooks in this document) to allow users to inject or override portions of the flow using higher-order functions operating on HammerTool in order to add, change, or suppress certain TCL generation behaviours in the program. For example, we include hooks to insert functions before or after a step (InsertPreStep/InsertPostStep), replace a step (ReplaceStep), resume before or after a

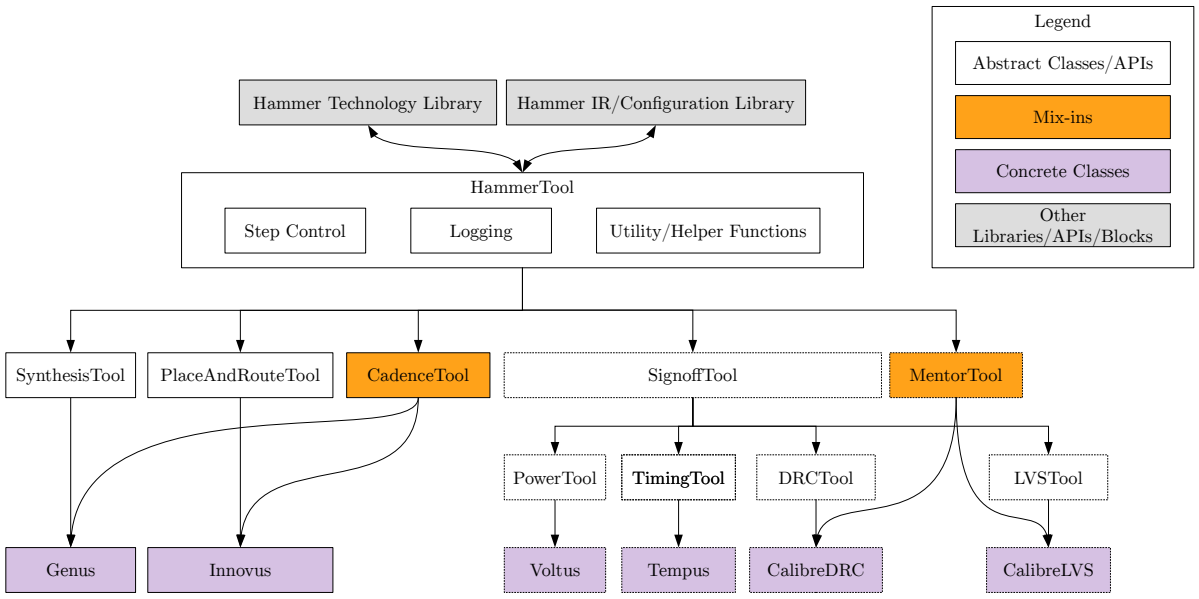

Figure 8: An overview of the Hammer tool abstraction (hammer-vlsi) in abstract APIs, mix-ins, concrete instances, and the relationships between them.

step (ResumePreStep/ResumePostStep) or pause the execution altogether.

After running, depending on the type of tool, each tool plugin populates a corresponding output snippet of Hammer IR containing the outputs from that run. For example, a synthesis run will populate outputs including the post-synthesis netlist and SDC constraints file.

```
@property
def steps(self) -> List[HammerToolStep]:
    return [
        self.init_design,
        self.floorplan_design,
        self.power_straps,
        self.place_opt_design,
        self.route_design,
        self.opt_design,
        self.write_design
    ]
```

CAD tool plugins are written as concrete classes which inherit from one or more of the abstract interfaces. Each plugin reads Hammer IR and emits the right TCL commands. Plugins can take full advantage of the the Hammer abstractions to minimize the effort required to write them. For example, Hammer comes with built-in functions for filtering from PDK libraries and processing them into the desired output format of the CAD tool (e.g. TCL or shell).

Finally, we can use Python's support for mixins[42] to "mix-in" common methods and functions for re-use between plugins. For example, many times CAD tools (say a synthesis

and a place-and-route tool) between the same vendor will share common startup/initialization commands (e.g. reading HDL, initializing libraries, etc). To help reduce code duplication and facilitate re-use between the two plugins, we can create a common class which contains helper functions for generating the common commands (e.g. `MyVendorTool`), and mix that into the actual tool classes (e.g. `MySynthesisTool` or `MyPlaceAndRouteTool`).

3.7 Technology Abstractions

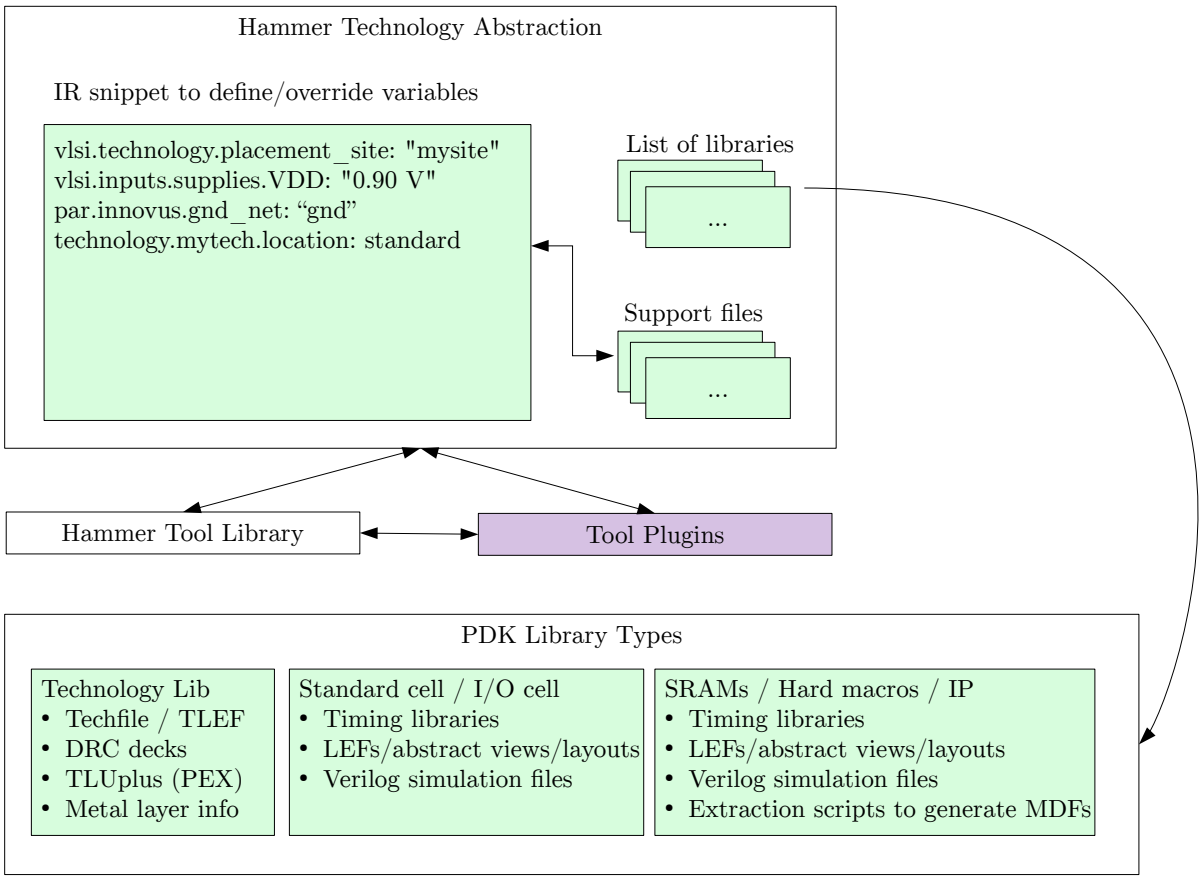

Figure 9: An overview of the three main components of the Hammer technology abstraction (hammer-tech).

A technology plugin in Hammer, implemented as a folder which the Hammer Technology library reads, consists of three main parts - 1) a `defaults.yml` or `defaults.json` snippet to define technology-specific IR settings, and to override any default tool or core (see Driver section) settings for this technology; 2) any support files necessary to support this technology properly - e.g. sometimes PDKs will have bugs or omissions which can be fixed with locally modified copies of files; 3) a list of libraries. The types of libraries supported include the technology itself (including technology LEFs, techfiles, DRC decks, etc) and standard cell/macro libraries, which include timing databases, Verilog simulation files, LEFs/Milkyway

databases, and some scripts to extract data e.g to generate MDF files (see below) for use in other parts of the Hammer methodology.

At the moment, we have ported to the following technology nodes to the Hammer Technology abstraction:

- ST 28nm FDSOI
- TSMC 16nm FinFET
- SAED32

Having independent technology plugins is essential to making Hammer portable. Technology PDKs are highly sensitive and often cannot be easily shared across organizational boundaries. By decoupling technology concerns from the rest of the system, we can isolate this particular area of concern using an open-source API, allowing us to be more portable across technologies, projects, and organizational boundaries. By creating a technology abstraction, we aim to make it possible to encode solutions to physical design problems more abstractly with more potential for re-use.

We organized our technology plugin this way in noticing that certain technology nodes required overriding certain CAD tool behaviours in certain ways or supplying sets of data to the CAD tools. We encode our list of libraries as a JSON array. We read this array into our technology library, which parses the file into Python data structures. The technology plugin instance is then attached by `hammer-vlsi` (the Hammer Tool Abstraction) and made available to CAD tool plugins to query. For example, we provide functions to filter and process libraries in the technology to make it easy to look for information from the technology abstraction (e.g. all LEFs from standard cell/macro libraries) and emit it in the appropriate manner for the CAD tool.

In addition to the above, the Hammer methodology features the macro description format (MDF), which is used to describe macros (type) and currently used most significantly for SRAMs (SRAM macros), though we plan to expand its use for other areas of backend VLSI (e.g. filler cells, I/O cells, etc). The description format is different for each type of macro - the most well-defined type of macro at the moment is the SRAM macro, which describes the width, depth, ports (names, read/write, masks, etc) of SRAMs. This format is currently used in the Hammer methodology by `MacroCompiler`, a pass which maps a generic Chisel/FIRRTL memory into a concrete implementation using a list of foundry SRAM macros. See the Agile RTL transformations and DSLs section for more about this.

3.8 Agile RTL transformations and DSLs

As outlined in the overview, in order to be truly agile (fast, automated feedback loops), we would like to automatically generate physical design constraints using the FIRRTL circuit and any associated annotations. A related project at Berkeley, CICL (Circuit Introspection and Command Language) is an API/DSL embedded in Scala for manipulating RTL circuits represented in FIRRTL. It consists of a selection language, backed by a graph representation (as opposed to an AST). This enables us to develop agile transformations and DSLs by making it easier to query and manipulate the circuit while generating physical design information.

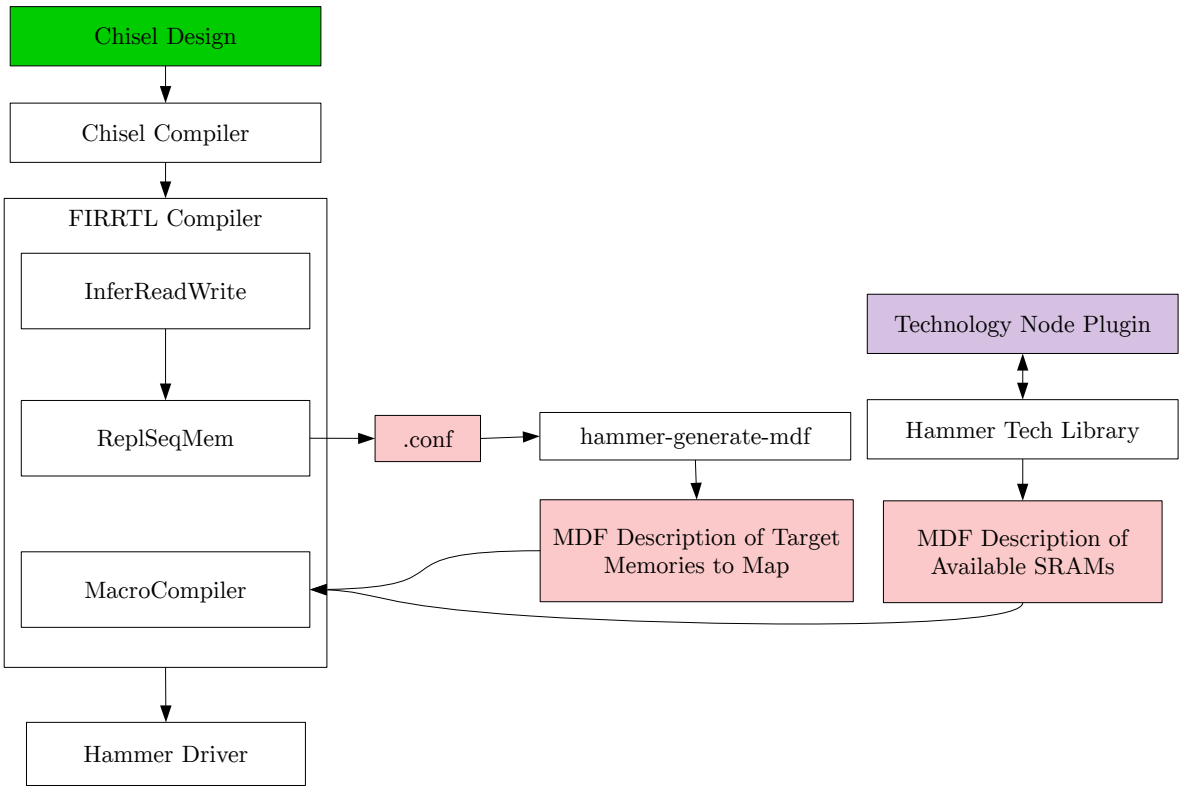

Figure 10: An example of using MDF to map abstract memories to technology-specific memories.

We want to tie the physical design constraint generation to the RTL generators due to powerful generators which can radically change RTL with parameter changes like rocket-chip, dsptools, etc.

For example, an operation that would be easily done in FIRRTL/CICL but tedious/non-reusable in TCL would be a transformation which takes half of an asynchronous FIFO and moves it to a different module with a different clock domain.

Should these transforms operate at the FIRRTL level? While there is theoretically nothing preventing you from writing Chisel code which generates Hammer IR, writing these transformations in FIRRTL allows you to address and manage circuit elements that didn't exist at the Chisel level yet, like memory macros generated by the MacroCompiler pass. In addition, not all designs are used in ASIC contexts, so having Chisel generators emit physical design constraints directly limits re-use.

MacroCompiler is a pass that takes FIRRTL with abstract memories. Using MDF information generated by Hammer from the technology plugin, it maps the abstract memory to an implementation using the foundry memories.

There are two main scenarios in terms of generating physical design information from Chisel/FIRRTL generators. The first is constraints that don't depend on technology/chip project information. In this case, we can encode these as FIRRTL annotations and use a simple FIRRTL pass to consume them and generate the appropriate Hammer IR. The second case involves constraints that depend on both the source RTL generator and technology/project

information. Examples of this include floorplanning (the same chip can be floorplanned in different ways to achieve different QoR), pad frame generation (depends on the technology as well as the pad/bump output), and clock/power gating constraints (requires insertion of power gating cells which depend on the technology).

As part of our efforts to explore the second scenario, we created an agile floorplan DSL to enable us to write floorplan generators corresponding to FIRRTL circuits which were portable and could emit physical design placement constraints in the Hammer IR.

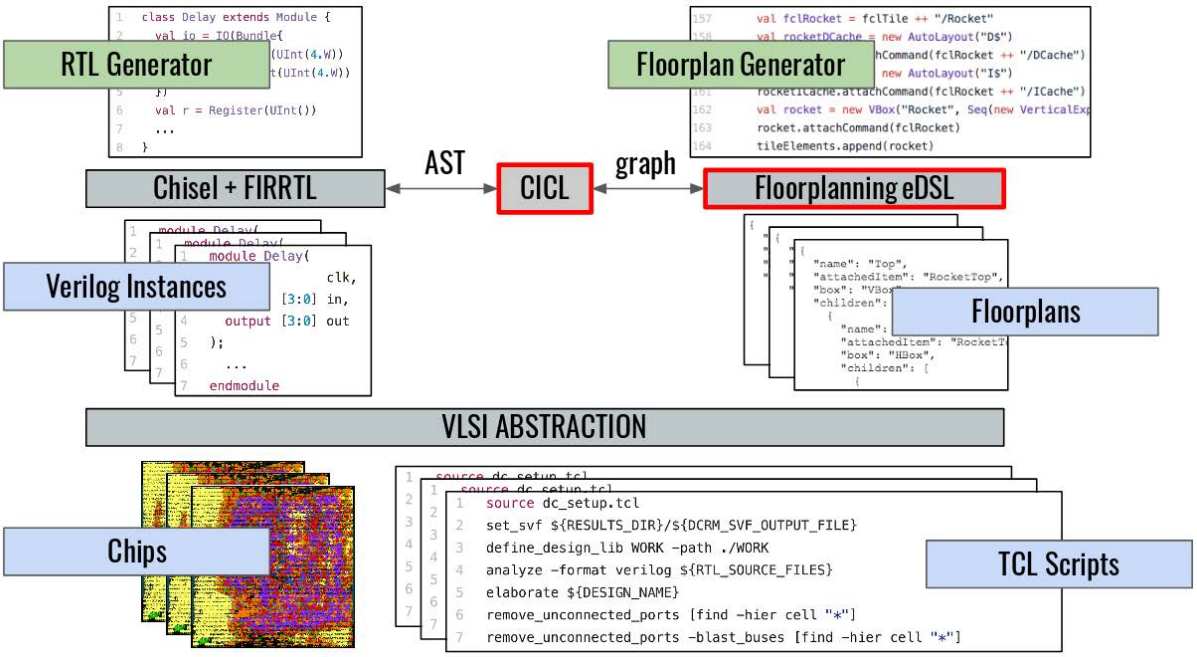

Figure 11: Floorplan generation - an example of a scenario where generation of physical design constraints requires both upstream knowledge from the RTL generator but also downstream technology/project information.

The floorplan DSL is designed to enable users (hardware engineers) to write re-usable layout generators corresponding to RTL. The floorplan DSL consists of three major portions - a geometry API, connections to CICL and FIRRTL for interfacing with the RTL, and a numeric solver to concretize the design with numbers to enable tapeout.

We provide a variety of geometry constructs to enable the construction of layouts. These constructs can be nested within one another and are independent of any numerical co-ordinates to enable re-use of layouts across different technologies and chip projects.

These constructs, which facilitates relative placement, include:

- HBox - tile the given elements horizontally.
- VBox - tile the given elements vertically.
- AutoLayout - do not specify any particular constraints to the backend tools for the module in this box.
- Expander - create a space/separate the given modules as much as possible.

- HardMacro - represents a hard macro, which has a fixed size but can vary in position.

Each geometry element can be attached to a module at the RTL level via a CICL Path, a selector for exactly one hardware component (in this case, a module). Not every geometry element is required to have a CICL Path. For example, a VBox could be attached to a module and have two sub-elements to floorplan parts of that module and an expander in between. In this case, the expander would not need a RTL module attached to it.

Notably, the RTL hierarchy does not need to correspond to the physical hierarchy. For example, consider a design with a Rocket in-order core with a Hwacha vector processing unit. Hwacha could exist as a submodule of Rocket while in the physical hierarchy, Hwacha and Rocket could be two parallel modules.[24]

In the DSL, geometry elements are connected to CICL's hardware module selectors via the attachPath() function.

Finally, in order to tape out a chip, a floorplan needs to be resolved into concrete numbers (position and dimensions). Since the exact co-ordinates for a particular module or layout will differ widely between different process nodes, we want to enable layout generators to be re-used as much as possible. In order to enable this, numbers are specified separately in a second stage as opposed to being required when the layout is created.

In the DSL, geometric elements have numbers entered to them via setDimensions() and setCoordinates(). We can instruct the DSL to resolve all the co-ordinates in the system (e.g. for visualization or export to backend tools) via resolveAll().

4 Evaluation

4.1 Sample Flows

```
git clone git@github.com:ucb-bar/hammer-examples.git
cd hammer-examples
git submodule update --init --recursive
# Edit sourceme.sh for the site setup
source sourceme.sh
```

Simple Adder Flow Given a simple 2-bit adder written in Verilog:

```
module adder (
input [1:0] a,
input [1:0] b,
output [1:0] c
);
assign c = a + b;
endmodule
```

[24] Synthesis might shuffle or flatten the hierarchy around. Currently it is the responsibility of the CAD tool plugin to ensure that floorplanning paths are valid and exist after synthesis.

Running synthesis and place-and-route using Hammer is trivial:

```
hammer-vlsi syn-par -o output.json -v adder.v --top adder
```

```
synthesis.inputs.input_files: ["SodorTile.v"]
synthesis.inputs.top_module: "SodorTile"
vlsi.inputs.clocks:
- name: "clock"
  period: "50 ns"
  uncertainty: "1 ns"
vlsi.inputs.placement_constraints:
- path: "SodorTile"
  type: "toplevel"
  x: 0
  y: 0
  width: 1500
  height: 1500
  margins:
    left: 100
    right: 100
    top: 100
    bottom: 100
- path: "SodorTile/core/d/csr"
  type: "placement"
  x: 600.0
  y: 600.0
  width: 30.0
  height: 200.0
- path: "SodorTile/memory/async_data/bleh"
  type: "hardmacro"
  x: 30.0
  y: 30.0
  width: 100.0
  height: 200.0



# Hierarchical example with manually specified placement constraints for each

synthesis.inputs.input_files: ["HierarchicalWriter.v"]

synthesis.inputs.top_module: "HierarchicalWriterTop"

vlsi.inputs.hierarchical.config_source: manual
vlsi.inputs.hierarchical.manual_modules:
```

```
- HierarchicalWriterTop:
  - HierarchicalWriter2
  HierarchicalWriter2:
  - HierarchicalWriter1

vlsi.inputs.hierarchical.manual_placement_constraints:
- HierarchicalWriterTop:
  - path: "HierarchicalWriterTop"
    type: "toplevel"
    x: 0
    y: 0
    width: 2600
    height: 1500
    margins:
      left: 50
      right: 50
      top: 50
      bottom: 100
  - path: "HierarchicalWriterTop/SRAM1RW512x32"
    type: "hardmacro"
    x: 1198
    y: 1200
    width: 204.684
    height: 247.541
  - path: "HierarchicalWriter2/HierarchicalWriter2"
    type: "hardmacro"
    x: 50
    y: 100
    width: 1200
    height: 1000
  - path: "HierarchicalWriter2/HierarchicalWriter2_1"
    type: "hardmacro"
    x: 1300
    y: 100
    width: 1200
    height: 1000
- HierarchicalWriter2:
  - path: "HierarchicalWriter2"
    type: "toplevel"
    x: 0
    y: 0
    width: 1200
    height: 1000
    margins:
      left: 50
```

```
      right: 100
      top: 100
      bottom: 100
  - path: "HierarchicalWriter2/SRAM1RW512x32"
    type: "hardmacro"
    x: 498
    y: 650
    width: 204.684
    height: 247.541
  - path: "HierarchicalWriter2/HierarchicalWriter1"
    type: "hardmacro"
    x: 50
    y: 100
    width: 500
    height: 500
  - path: "HierarchicalWriter2/HierarchicalWriter1_1"
    type: "hardmacro"
    x: 600
    y: 100
    width: 500
    height: 500
- HierarchicalWriter1:
  - path: "HierarchicalWriter1"
    type: "toplevel"
    x: 0
    y: 0
    width: 500
    height: 500
    margins:
      left: 50
      right: 50
      top: 50
      bottom: 50
  - path: "HierarchicalWriter1/SRAM1RW512x32"
    type: "hardmacro"
    x: 60
    y: 60
    width: 204.684
    height: 247.541

vlsi.inputs.clocks:
- name: "clock"
  period: "50 ns"
uncertainty: "1 ns"
```

Porting to a Different CAD Tool Vendor Given two snippets (`use_synopsys.json` and `use_cadence.json`):

`use_cadence.json`

```
{
  "vlsi.core.synthesis_tool": "genus",
  "vlsi.core.par_tool": "innovus",
  ...
}
```

`use_synopsys.json`

```
{
  "vlsi.core.synthesis_tool": "dc",
  "vlsi.core.par_tool": "icc",
  ...
}
```

Switching is as easy as going from:

```
hammer-vlsi syn-par -o output.json -v adder.v --top adder \\
  -e use_synopsys.json
```

To:

```
hammer-vlsi syn-par -o output.json -v adder.v --top adder \\
  -e use_cadence.json
```

Porting to a Different Technology Create a JSON/YAML IR snippet to reference the appropriate technology (YAML example below) and as before, use `-e` to point to the new technology.

`use_mytech.yml`

```
vlsi.core.technology: my_tech
vlsi.core.technology_path: ["path_to_my_tech"]
vlsi.core.technology_path_meta: append
```

```
hammer-vlsi syn-par -o output.json -v adder.v --top adder -e use_mytech.yml
```

```
export HAMMER_ENVIRONMENT_CONFIGS="/path/to/file/with/local/settings.json"
```

4.2 EAGLE

EAGLE is an 8-core (Rocket in-order core+ Hwacha vector unit), 4-cluster SoC with 5 SerDes lanes in TSMC 16nm being taped out at the BWRC and ADEPT lab at UC Berkeley. Our

preliminary estimates for QoR include power consumption in the range of 6-7 watts, area of 5mm x 5mm, and a maximum clock rate of 980 MHz.

A successor to Hurricane-2[25] and CRAFT in terms of design methodology, EAGLE provided us with an opportunity to explore the benefits of the new Hammer methodology. We took the information encoded in previous 16nm CRAFT tapeouts and used it to write a new, re-usable 16nm technology plugin for Hammer. This plugin is in fact being used by multiple 16nm projects here at Berkeley, highlighting its re-usability.

EAGLE (`eagle-vlsi`) uses the CLIDriver abstraction to modify the default Hammer flow abstraction, since we weren't creating an entirely custom application/generator. Most of the customizations are via TCL hooks, which allowed us to incrementally use Hammer, increasing adoption and usability, especially while hooks were being added. For example, we used TCL hooks to implement a hierarchical flow while the Hammer hierarchical API was still being developed. Likewise, we currently use TCL hooks for extra VLSI/physical design features, like RDL routing, filler cells, tap cells, etc. This is consistent with our philosophy of incremental adoption. In fact, due to its open source nature, certain parts of Hammer used by EAGLE designers were found to be easier to use than competing commercial tools like flowtool.

Note that neither the hammer nor hammer-cad-plugins repo was forked during this tapeout, a first after many previous tapeouts involved wholesale forking/copying/modifying directly upstream flows and generators. For example, the Hurricane-2 tapeout repo was a fork of the rocket-chip generator with a heavily customized/mutated Synopsys RM copied/merged into it.

4.3 EE194/290C

EE194/290C is a special topics class at UC Berkeley designed to teach advanced undergraduates who have never taped out before a hands-on experience in taping out a chip. Last iteration of the course, we used Synopsys tools and a traditional-style flow[26]. As in the flow for EAGLE, we never forked hammer or hammer-cad-plugins.

To give an idea of how much time we saved using re-usable libraries/frameworks/abstractions:

- Only 3 hours to initial GDS flow, ST28, first time using Cadence flows. Some debugging due to LEF files. No SRAMs yet.
- About 2 hours to investigate results and create power grid.
- About 5 hours (mostly editing paths in .tech.json files) to porting 10-layer 2.7 PDK to 8-layer 2.9 PDK (modulo the foundry dropping .lib files in some libraries)
- 3 hours porting new SRAMs and generating them with MacroCompiler.

[25] Unfortunately no publication is available yet at the time of writing.

[26] We attempted to use PLSI but it devolved into a traditional flow because it was difficult to customize the flow without effectively forking it into a separate repo.

- 5-6 hours to understand + implement I/O cells and implement usage in Innovus.

5 Related Work

- PLSIis the closest work to this one. However, it is designed framework-style and in low-level programming languages, hindering adoption, re-use, and maintenance.
- Flowtoolis a set of flow management scripts written in TCL for Cadence tools. It is not portable across CAD tool vendors, mixes in concerns of CAD tool vendors, technologies, and project, and as with PLSI, is low-level.
- `vlsi_mem_gen` et al are Python scripts used to generate Verilog for various physical design problems (e.g. SRAM mapping, I/O cell generation, etc) used at Berkeley. While they work they are stringly-typed, unsafe, and not reusable (every chip project has its own copy of this script with very subtle modification that prevent re-use).
- Spongepaintis a procedural layout generator written in Java. It does not use a front-end HDL like Verilog making re-use problematic, and is closer to BAG[31] in style as a direct-to-layout generator as opposed to our solution which leverages the power of modern place-and-route tools by generating constraints for them.
- Jackhammeris a DSE framework which solves a parallel/orthogonal problem and theoretically could plug into Hammer to run the backend but is parallel.
- https://pdfs.semanticscholar.org/4e45/d145742228596753fee7fb440818c102390a.pdf seems to be about 3D ICs
- http://www.inf.pucrs.br/~calazans/publications/2015-DSD_ACSD.pdf only applicable to self-timed circuit, not generic, not re-usable
- State of the art in industry: many companies have internal flows that aren't published. They suffer from the problems described in the background section of this work.

6 Lessons Learned

- To build useful tools, listen to users to understand their needs and use-cases, but also don't hesitate to introduce new tools, languages, or paradigms to solve problems.
- Dogfooding (the act of using the software one develops) goes a long way to ensuring that tools are usable in the real world.

- Think about what environments the tool will be deployed in, and anticipate logistics (e.g. NDAs, licensing, etc) problems and develop solutions (e.g. abstractions) for them to increase portability.
- Rocket is hard to use because it doesn't have textual descriptions of its building blocks, forcing people to read the code. Reading code is harder and slower than writing code. If writing the code yourself is faster, we just missed the point of agile re-use.
- Likewise, existing IP blocks used in hardware design[36] are hard to use because they are unparametrizable and hard to customize to the target/project environment, inviting the temptation to just write the code yourself again.
- Examples and templates go a long way to helping push adoption - think of chisel-template, project-template, or hwacha-template.
- Non immutable data structures suck - https://github.com/ucb-bar/hammer/commit/256eef15e2d558debe84dfed3bd4291fd2364a32

7 Future Directions

- Simulation - we can use Hammer to set up the flow and environment for that; we just haven't written the infrastructure to do so yet.
- Incorporate input from analog designers to achieve a layout that is most optimal for their designs as well - if we have a Chisel/Hammer (digital/physical design) generator and a BAG (analog) generator, we could write a design space exploration/optimization engine which starts from a seed set of parameters, runs both generators, integrates them, evaluates the resultant QoR, and adjusts the parameters accordingly to increase QoR. There is potential to apply some optimization/ML in this scenario.
- Alternatively, the generators could feature a two-phase Diplomacy-like API where in the first pass part of the API, we obtain an acceptable range of values for a set of certain parameters from each part of the generator, and the call the second pass part to generate a concrete instance using both generators. This is similar to the above scenario but with an extra API queries what is acceptable to both generators.
- Design space exploration engine (using ML or traditional optimization) to automatically set parameters to increase agility.
- Expand the number/variety of Hammer project templates/examples.
- Integration with analog generators like BAG.
- Integration with formal verification.

- Extend the work to support FPGA flows, or co-generation of FPGA and ASIC flows in order to make it easier to do agile pre-tapeout simulations of ASICs. This could be accomplished using FIRRTL/CICL passes as well as interfacing with MIDAS.
- Explore type-safe aspects/Hammer IR generation at the Chisel level.
- Integration for push-button (GUI) generators.
- More CAD tool knobs.
- More hammer-vlsi plugins for more tools.

8 Conclusion

Hardware design faces a critical crossroads - increased demand for performance from the end of Moore's law meeting the inefficient design practices common in the hardware design scene today. We recognize that current design methodology of manually writing VLSI flow scripts that combine information about tools, technologies, and the RTL design limits productivity and portability. This is beecause digital physical design effort is not shared and re-used across different tools/technologies/RTL designs.

As such, we created the Hammer methodology, a modular platform consisting of 1) libraries to abstract tools and technologies; 2) a physical design IR for interchange of physical design information between upstream generators and backend VLSI tools that is human-writable (e.g. in YAML with comments) as well as programmatically generatable; 3) drivers and shell wrappers to make it easy to get started in using the Hammer methodology.

While there remains more work to do to improve Hammer and add more features, we explicitly design Hammer to enable partial and incremental adoption via mechanisms like TCL hooks. This allows expert users to directly write TCL, while continuing to use Hammer to support their existing features without requiring an all-or-nothing migration. We aim to build a tool useful to a wide audience, including but not limited to computer architects, RTL engineers, and physical designers.

Similar to how the SHARE society of the 1950s aimed to reduce redundant effort among software developers, we aim to help everyone share the load and bridge the gap between architectural exploration and taping out by making it easier to tape out, while simultaneously reducing the amount of engineer hours spent wasted due to inefficient design practices.

Acknowledgements

This work was funded in part by the DARPA CRAFT program (HR0011-16-C-0052).

Firstly, I would like to thank my primary advisor, **Professor Jonathan Bachrach**, for being patient in helping me to find my research path and in introducing me to many great collaborators, many of whom are listed below. I would also like to thank my other faculty

advisors, **Professor Elad Alon** and **Professor Krste Asanović** for their mentorship and enthusiasm in pursuing this project.

Fellow ADEPT colleagues **Adam Izraelevitz** and **Colin Schmidt** were trailblazers in forging the path towards agile hardware design, yet also managed to squeeze time in their busy schedules to help me grow as a person both technically and non-technically. Our numerous fruitful conversations together have left a lasting impact on my collaboration proficiency.

John Wright joined **Adam Izraelevitz**, **Colin Schmidt**, and me in many thought-provoking discussions, thereby providing valuable user feedback while the HAMMER methodology was still in its infancy.

Professor Kristofer Pister's visionary mindset helped to pave a crucial pathway within the department, providing an outlet for undergraduates (including myself) to tape out. Although his research interests do not directly intersect with mine, he has provided valuable guidance on how to maintain a bird's eye view of my research goals.

I would also like to thank my graduate student colleagues in the BAR group, ADEPT lab, and the Berkeley Wireless Research Center (BWRC). In particular, I would like to thank my colleagues who helped read drafts - **Steven Bailey**, **Jim Lawson**, **Adam Izraelevitz**, **Nathan Pemberton**, and **Richard Lin**.

The staff at the BWRC helped with many very important logistical details ranging from setting up accounts to organizing our semester-end retreats - the fabulous **Candy Corpus**, **Ria Briggs**, and **Tami Chouteau**. At the ADEPT lab, staff scientists **Jim Lawson**, **Chick Markley**, and system administrator **Kostadin Ilov**, as well as the BWRC sysadmins (**James Dunn** and **Gregory Pearson**), as well as staff R&D engineer **Brian Richards** helped to maintain crucial equipment, without which the meat of this project would not have been possible.

My parents, **ChinJohn Wang** and **Xiang Yang** provided me with the opportunity and means to attend this great institution. I would also like to thank my brother **Chency Wang** for providing moral support.

Last but not least, I would like to thank **Rachel Zoll** for her support throughout this entire process.

Appendices

A Reproducibility

Open source code in this thesis is available on Github (some repositories may be NDA-restricted, private, or moved to other URLs; contact the author for access):

- Hammer
  - Repository: https://github.com/ucb-bar/hammer
  - Branch: master
  - Commit: e99efd9d2056677e0167a8dcc634dbdd16016803
- Hammer CAD Tool Plugins
  - Repository: https://github.com/ucb-bar/hammer-cad-plugins
  - Branch: master
  - Commit: 85547c9a1cac351e9d88870ec6b04d0257f54d0b
- CICL and Floorplanning
  - Repository: https://github.com/ucb-bar/fcl-floorplan
  - Branch: master
  - Commit: 419ed288f4f24284f833b80bfd26f67fcbaa6b06
- Barstools (Agile FIRRTL/CICL passes for tapeouts)
  - Repository: https://github.com/ucb-bar/barstools
  - Branch: master
  - Commit: 93bf7895bee4fe866ede244e91da9514bb321087

B Hammer Tool and Driver API Documentation

CHAPTER

ONE

HAMMER_VLSI MODULE

class hammer_vlsi.**CLIDriver** → None

Bases: object

Helper class for projects to easily write/customize a CLI driver for hammer without needing to rewrite/copy all the argparse and plumbing.

action_map() → typing.Dict[str, typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]]

Return the mapping of valid actions -> functions for each action of the command-line driver.

all_hierarchical_actions

Return a list of hierarchical actions if the given project configuration is a hierarchical design. Set when the driver is first created in args_to_driver. Create syn/synthesis-[block], par-[block], and /syn_par-[block].

Returns Dictionary of actions to use (could be empty).

args_to_driver(*args: dict*, *default_options: typing.Union[hammer_driver.HammerDriverOptions, NoneType] = None*) → typing.Tuple[hammer_driver.HammerDriver, typing.List[str]]

Parse command line arguments and environment variables for the command line front-end to hammer-vlsi.

Returns HammerDriver and a list of errors.

create_action(*action_type: str*, *extra_hooks: typing.Union[typing.List[hammer_vlsi_impl.HammerToolHookAction], NoneType]*, *pre_action_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None*, *post_load_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None*, *post_run_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None*) → typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]

Create an action function for the action_map.

Parameters

- **action_type** – Either "syn"/"synthesis" or "par"
- **extra_hooks** – List of hooks to pass to the run function.
- **pre_action_func** – Optional function to call before doing anything.
- **post_load_func** – Optional function to call after loading the tool.
- **post_run_func** – Optional function to call after running the tool.

Returns Action function.

create_par_action(*custom_hooks: typing.List[hammer_vlsi_impl.HammerToolHookAction], pre_action_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None, post_load_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None, post_run_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None*) → typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]

create_synthesis_action(*custom_hooks: typing.List[hammer_vlsi_impl.HammerToolHookAction], pre_action_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None, post_load_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None, post_run_func: typing.Union[typing.Callable[[hammer_driver.HammerDriver], NoneType], NoneType] = None*) → typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]

create_synthesis_par_action(*synthesis_action: typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]], par_action: typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]*) → typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]

Create a parameterizable synthesis_par action for the CLIDriver.

Parameters

- **synthesis_action** – synthesis action
- **par_action** – par action

Returns Custom synthesis_par action

get_extra_hierarchical_par_hooks() → typing.Dict[str, typing.List[hammer_vlsi_impl.HammerToolHookAction]]

Return a list of extra hierarchical place and route hooks in this project. To be overridden by subclasses.

Returns Dictionary of (module name, list of hooks)

get_extra_hierarchical_synthesis_hooks() → typing.Dict[str, typing.List[hammer_vlsi_impl.HammerToolHookAction]]

Return a list of extra hierarchical synthesis hooks in this project. To be overridden by subclasses.

Returns Dictionary of (module name, list of hooks)

get_extra_par_hooks() → typing.List[hammer_vlsi_impl.HammerToolHookAction]

Return a list of extra place and route hooks in this project. To be overridden by subclasses.

get_extra_synthesis_hooks() → typing.List[hammer_vlsi_impl.HammerToolHookAction]

Return a list of extra synthesis hooks in this project. To be overridden by subclasses.

get_hierarchical_par_action(*module: str*) → typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]

Get the action associated with hierarchical par for the given module (in hierarchical flows).

get_hierarchical_synthesis_action(*module: str*) → typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]
Get the action associated with hierarchical synthesis for the given module (in hierarchical flows).

get_hierarchical_synthesis_par_action(*module: str*) → typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]
Get the action associated with hierarchical syn_par for the given module (in hierarchical flows).

main() → None
Main function to call from your entry point script. Parses command line arguments. Example: >>> if __name__ == '__main__': >>> CLIDriver().main()

run_main_parsed(*args: dict*) → int
Given a parsed dictionary of arguments, find and run the given action.

Returns Return code (0 for success)

set_hierarchical_par_action(*module: str, action: typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]*) → None
Set the action associated with hierarchical par for the given module (in hierarchical flows).

set_hierarchical_synthesis_action(*module: str, action: typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]*) → None
Set the action associated with hierarchical synthesis for the given module (in hierarchical flows).

set_hierarchical_synthesis_par_action(*module: str, action: typing.Callable[[hammer_driver.HammerDriver, typing.Callable[str, NoneType]], typing.Union[dict, NoneType]]*) → None
Set the action associated with hierarchical syn_par for the given module (in hierarchical flows).

synthesis_to_par_action(*driver: hammer_driver.HammerDriver, append_error_func: typing.Callable[str, NoneType]*) → typing.Union[dict, NoneType]
Create a config to run the output.

valid_actions() → typing.List[str]
Get the list of valid actions for the command-line driver.

class hammer_vlsi.**CadenceTool**
Bases: hammer_vlsi_impl.HasSDCSupport, hammer_vlsi_impl.HammerTool

Mix-in trait with functions useful for Cadence-based tools.

config_dirs

env_vars
Get the list of environment variables required for this tool. Note to subclasses: remember to include variables from super().env_vars!

generate_mmmc_script() → str
Output for the mmmc.tcl script. Innovus (init_design) requires that the timing script be placed in a separate file.

Returns Contents of the mmmc script.

get_liberty_libs() → str
Helper function to get the list of ASCII liberty files in space separated format.

Returns List of lib files separated by spaces

get_mmmc_libs(*corner: hammer_vlsi_impl.MMMCCorner*) → str

get_mmmc_qrc(*corner: hammer_vlsi_impl.MMMCCorner*) → str

get_qrc_tech() → str
Helper function to get the list of rc corner tech files in space separated format.

Returns List of qrc tech files separated by spaces

class hammer_vlsi.**Callable**
Bases: collections.abc.Callable

Callable type; Callable[[int], str] is a function of (int) -> str.

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types or ellipsis; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments, such function types are rarely used as callback types.

class hammer_vlsi.**ClockPort**(*name*, *period*, *port*, *uncertainty*)
Bases: tuple

name
Alias for field number 0

period
Alias for field number 1

port
Alias for field number 2

uncertainty
Alias for field number 3

class hammer_vlsi.**FullMessage**(*message*, *level*, *context*)
Bases: tuple

context
Alias for field number 2

level
Alias for field number 1

message
Alias for field number 0

class hammer_vlsi.**HammerDriver**(*options: hammer_driver.HammerDriverOptions*, *extra_project_config: dict = {}*) → None
Bases: object

static generate_par_inputs_from_synthesis(*config_in: dict*) → dict
Generate the appropriate inputs for running place-and-route from the outputs of synthesis run.

static get_default_driver_options() → hammer_driver.HammerDriverOptions
Get default driver options.

get_hierarchical_settings() → typing.List[typing.Tuple[str, dict]]
Read settings from the database, determine leaf/hierarchical modules, an order of execution, and return an

ordered list (from leaf to top) of modules and associated config snippets needed to run syn+par for that module hierarchically.

Returns List of tuples of (module name, config snippet)

load_par_tool(*run_dir: str = ''*) → bool
Load the place and route tool based on the given database.

Parameters **run_dir** – Directory to use for the tool run_dir. Defaults to the run_dir passed in the HammerDriver constructor.

load_synthesis_tool(*run_dir: str = ''*) → bool
Load the synthesis tool based on the given database.

Parameters **run_dir** – Directory to use for the tool run_dir. Defaults to the run_dir passed in the HammerDriver constructor.

Returns True if synthesis tool loading was successful, False otherwise.

load_technology(*cache_dir: str = ''*) → None

project_config

run_par(*hook_actions: typing.Union[typing.List[hammer_vlsi_impl.HammerToolHookAction], NoneType] = None, force_override: bool = False*) → typing.Tuple[bool, dict]
Run place and route based on the given database.

run_synthesis(*hook_actions: typing.Union[typing.List[hammer_vlsi_impl.HammerToolHookAction], NoneType] = None, force_override: bool = False*) → typing.Tuple[bool, dict]
Run synthesis based on the given database.

Parameters

- **hook_actions** – List of hook actions, or leave as None to use the hooks sets in set_synthesis_hooks. Hooks from set_synthesis_hooks, if present, will be appended afterwards.
- **force_override** – Set to true to overwrite instead of append.

Returns Tuple of (success, output config dict)

set_post_custom_par_tool_hooks(*hooks: typing.List[hammer_vlsi_impl.HammerToolHookAction]*) → None
Set the extra list of hooks used for control flow (resume/pause) in run_par. They will run after main/hook_actions.

Parameters **hooks** – Hooks to run

set_post_custom_syn_tool_hooks(*hooks: typing.List[hammer_vlsi_impl.HammerToolHookAction]*) → None
Set the extra list of hooks used for control flow (resume/pause) in run_synthesis. They will run after main/hook_actions.

Parameters **hooks** – Hooks to run

update_project_configs(*project_configs: typing.List[dict]*) → None
Update the project configs in the driver and database.

update_tool_configs() → None
Calls self.database.update_tools with self.tool_configs as a list.

class hammer_vlsi.**HammerDriverOptions**(*environment_configs*, *project_configs*, *log_file*, *obj_dir*)
Bases: tuple

environment_configs
Alias for field number 0

log_file
Alias for field number 2

obj_dir
Alias for field number 3

project_configs
Alias for field number 1

class hammer_vlsi.**HammerPlaceAndRouteTool**
Bases: hammer_vlsi_impl.HammerTool

export_config_outputs() → typing.Dict[str, typing.Any]

fill_outputs() → bool

input_files
Get the input post-synthesis netlist files.

Returns The input post-synthesis netlist files.

output_ilms
Get the (optional) output ILM information for hierarchical mode.

Returns The (optional) output ILM information for hierarchical mode.

post_synth_sdc
Get the input post-synthesis SDC constraint file.

Returns The input post-synthesis SDC constraint file.

top_module
Get the top RTL module.

Returns The top RTL module.

hammer_vlsi.**HammerStepFunction**
alias of *Callable*

class hammer_vlsi.**HammerSynthesisTool**
Bases: hammer_vlsi_impl.HammerTool

export_config_outputs() → typing.Dict[str, typing.Any]

fill_outputs() → bool

input_files
Get the input collection of source RTL files (e.g. *.v).

Returns The input collection of source RTL files (e.g. *.v).

output_files
Get the output collection of mapped (post-synthesis) RTL files.

Returns The output collection of mapped (post-synthesis) RTL files.

output_sdc
Get the (optional) output post-synthesis SDC constraints file.

Returns The (optional) output post-synthesis SDC constraints file.

top_module
Get the top-level module.

Returns The top-level module.

class hammer_vlsi.**HammerTool**

Bases: object

static append_contents_to_path(*content_to_append: str*, *target_path: str*) → None

Append the given contents to the file located at target_path, if target_path is not empty.

Parameters

- **content_to_append** – Content to append.
- **target_path** – Where to append the content.

attr_getter(*key: str*, *default: typing.Any*) → typing.Any

Helper function for implementing the getter of a property with a default. If default is None, then raise a AttributeError.

attr_setter(*key: str*, *value: typing.Any*) → None

Helper function for implementing the setter of a property with a default.

check_duplicates(*lst: typing.List[hammer_vlsi_impl.HammerToolStep]*) → typing.Tuple[bool, typing.Set[str]]

Check that no two steps have the same name.

check_input_files(*extensions: typing.List[str]*) → bool

Verify that input files exist and have the specified extensions.

Parameters **extensions** – List of extensions e.g. [".v", ".sv"]

Returns True if all files exist and have the specified extensions.

config_dirs

List of folders where (default) configs can live. Defaults to self.tool_dir.

Returns List of default config folders.

create_enter_script(*enter_script_location: str = ''*, *raw: bool = False*) → None

Create the enter script inside the rundir which can be used to create an interactive environment with all the same variables used to launch this tool.

Parameters

- **enter_script_location** – Location to create the enter script. Defaults to self.run_dir + "/enter"
- **raw** – Emit the raw string without shell escaping (without quotes!!!)

static create_nonempty_check(*description: str*) → typing.Callable[typing.List[str], typing.List[str]]

do_between_steps(*prev: hammer_vlsi_impl.HammerToolStep*, *next: hammer_vlsi_impl.HammerToolStep*) → bool

Function to run after the list of steps executes. Does not include pause hooks. Intended to be overridden by subclasses.

Parameters

- **prev** – The step that just finished
- **next** – The next step about to run.

Returns True if successful, False otherwise.

do_post_steps() → bool

Function to run after the list of steps executes. Intended to be overridden by subclasses.

Returns True if successful, False otherwise.

do_pre_steps(*first_step: hammer_vlsi_impl.HammerToolStep*) → bool
Function to run before the list of steps executes. Intended to be overridden by subclasses.

Parameters **first_step** – First step to be taken.

Returns True if successful, False otherwise.

dump_database() → str
Dump the current database JSON in a temporary file in the run_dir and return the path.

env_vars
Get the list of environment variables required for this tool. Note to subclasses: remember to include variables from super().env_vars!

Returns Mapping of environment variable -> contents of said variable.

export_config_outputs() → typing.Dict[str, typing.Any]
Export the outputs of this tool to a config.

Returns Config dictionary of the outputs of this tool.

fill_outputs() → bool
Fill the outputs of the tool. Note: if you override this, remember to call the superclass method too!

Returns True if successful, False otherwise.

filter_and_select_libs(*lib_filters: typing.List[typing.Callable[abc.Library, bool]] = [], sort_func: typing.Union[typing.Callable[[abc.Library], typing.Union[numbers.Number, str, tuple]], NoneType] = None, extraction_func: typing.Callable[abc.Library, typing.List[str]] = None, extra_funcs: typing.List[typing.Callable[str, str]] = []*) → typing.List[str]
Generate a list by filtering the list of libraries and selecting some parts of it.

Parameters

- **lib_filters** – Filters to filter the list of libraries before selecting desired results from them. e.g. remove libraries of the wrong type
- **sort_func** – Sort function to re-order the resultant components. e.g. put stdcell libraries before any other libraries
- **extraction_func** – Function to call to extract the desired component of the lib. e.g. turns the library into the ".lib" file corresponding to that library
- **extra_funcs** – List of extra functions to call before wrapping them in the arg prefixes.

Returns List generated from list of libraries

filter_for_mmmc(*voltage*, *temp*) → typing.Callable[abc.Library, bool]
Selecting libraries that match given temp and voltage.

filter_for_supplies(*lib: abc.Library*) → bool
Function to help filter a list of libraries to find libraries which have matching supplies. Will also use libraries with no supplies annotation.

Parameters **lib** – Library to check

Returns True if the supplies of this library match the inputs for this run, False otherwise.

get_clock_ports() → typing.List[hammer_vlsi_impl.ClockPort]
Get the clock ports of the top-level module, as specified in vlsi.inputs.clocks.

get_config() → typing.List[dict]
Get the config for this tool.

get_input_ilms() → typing.List[hammer_vlsi_impl.ILMStruct]
Get a list of input ILM modules for hierarchical mode.

get_mmmc_corners() → typing.List[hammer_vlsi_impl.MMMCCorner]
Get a list of MMMC corners as specified in the config.

get_output_load_constraints() → typing.List[hammer_vlsi_impl.OutputLoadConstraint]
Get a list of output load constraints as specified in the config.

get_placement_constraints() → typing.List[hammer_vlsi_impl.PlacementConstraint]
Get a list of placement constraints as specified in the config.

get_setting(*key: str*, *nullvalue: typing.Union[str*, *NoneType] = None*) → typing.Any
Get a particular setting from the database.

Parameters

- **key** – Key of the setting to receive.
- **nullvalue** – Value to return in case of null (leave as None to use the default).

hierarchical_mode
Input files for this tool library. The exact nature of the files will depend on the type of library.

input_files
Input files for this tool library. The exact nature of the files will depend on the type of library.

lef_filter
Select LEF files for physical layout.

liberty_lib_filter
Selecting ASCII liberty (.lib) libraries. Prefers CCS if available; picks NLDM as a fallback.

logger
Get the logger for this tool.

static make_check_isdir(*description: str = 'Path'*) → typing.Callable[str, str]
Utility function to generate functions which check whether a path exists.

static make_check_isfile(*description: str = 'File'*) → typing.Callable[str, str]
Utility function to generate functions which check whether a path exists.

static make_from_to_hooks(*from_step: typing.Union[str, NoneType] = None*, *to_step: typing.Union[str, NoneType] = None*) → typing.List[hammer_vlsi_impl.HammerToolHookAction]
Helper function to create a HammerToolHookAction list which will run from and to the given steps, inclusive.

Parameters

- **from_step** – Run from the given step, inclusive. Leave as None to resume from the beginning.
- **to_step** – Run to the given step, inclusive. Leave as None to run to the end.

Returns HammerToolHookAction list for running from and to the given steps, inclusive.

static make_insertion_hook(*step: str, location: hammer_vlsi_impl.HookLocation, func: typing.Callable[_ForwardRef('HammerTool'), bool]*) → hammer_vlsi_impl.HammerToolHookAction
Create a hook action is inserted relative to the given step.

static make_pause_function() → typing.Callable[_ForwardRef('HammerTool'), bool]
Get a step function which will stop the execution of the tool.

static make_post_insertion_hook(*step: str, func: typing.Callable[_ForwardRef('HammerTool'), bool]*) → hammer_vlsi_impl.HammerToolHookAction
Create a hook action is inserted after the given step.

static make_post_pause_hook(*step: str*) → hammer_vlsi_impl.HammerToolHookAction
Create pause before the execution of the given step.

static make_post_resume_hook(*step: str*) → hammer_vlsi_impl.HammerToolHookAction
Resume after the given step. Note that only one resume hook may be present.

static make_pre_insertion_hook(*step: str, func: typing.Callable[_ForwardRef('HammerTool'), bool]*) → hammer_vlsi_impl.HammerToolHookAction
Create a hook action is inserted prior to the given step.

static make_pre_pause_hook(*step: str*) → hammer_vlsi_impl.HammerToolHookAction
Create pause before the execution of the given step.

static make_pre_resume_hook(*step: str*) → hammer_vlsi_impl.HammerToolHookAction
Resume before the given step. Note that only one resume hook may be present.

static make_removal_hook(*step: str*) → hammer_vlsi_impl.HammerToolHookAction
Helper function to remove a step by replacing it with an empty step.

Returns Hook action which replaces the given step.

static make_replacement_hook(*step: str, func: typing.Callable[_ForwardRef('HammerTool'), bool]*) → hammer_vlsi_impl.HammerToolHookAction
Create a hook action which replaces an existing step.

Returns Hook action which replaces the given step.

static make_resume_hook(*step: str, location: hammer_vlsi_impl.HookLocation*) → hammer_vlsi_impl.HammerToolHookAction
Create a hook action is inserted relative to the given step.

static make_step_from_function(*func: typing.Callable[_ForwardRef('HammerTool'), bool], name: str = ''*) → hammer_vlsi_impl.HammerToolStep
Create a HammerToolStep from a function.

Parameters

- **func** – Class function for the given substep
- **name** – Name of the hook. If unspecified, defaults to func.__name__.

Returns A HammerToolStep defining this step.

static make_step_from_method(*func: typing.Callable[bool], name: str = ''*) → hammer_vlsi_impl.HammerToolStep
Create a HammerToolStep from a method.

Parameters

- **func** – Method for the given substep (e.g. self.elaborate)
- **name** – Name of the hook. If unspecified, defaults to func.__name__.

Returns A HammerToolStep defining this step.

static make_steps_from_methods(*funcs: typing.List[typing.Callable[bool]]*) → typing.List[hammer_vlsi_impl.HammerToolStep]
Create a series of HammerToolStep from the given list of bound methods.

Parameters **funcs** – List of bound methods (e.g. [self.step1, self.step2])

Returns List of HammerToolSteps

milkyway_lib_dir_filter

milkyway_techfile_filter
Select milkyway techfiles.

name
Short name of the tool library. Typically the folder name (e.g. "dc", "yosys", etc).

Returns Short name of the tool library.

process_library_filter (*pre_filts: typing.List[typing.Callable[abc.Library, bool]], filt: hammer_vlsi_impl.LibraryFilter, output_func: typing.Callable[[str, hammer_vlsi_impl.LibraryFilter], typing.List[str]], must_exist: bool = True*) → typing.List[str]
Process the given library filter and return a list of items from that library filter with any extra post-processing.

- Get a list of lib items
- Run any extra_post_filter_funcs (if needed)
- For every lib item in each lib items, run output_func

Parameters

- **pre_filts** – List of functions with which to pre-filter the libraries. Each function must return true in order for this library to be used.
- **filt** – LibraryFilter to check against the list.
- **output_func** – Function which processes the outputs, taking in the filtered lib and the library filter which generated it.
- **must_exist** – Must each library item actually exist? Default: True (yes, they must exist)

Returns Resultant items from the filter and post-processed. (e.g. –timing foo.db –timing bar.db)

qrc_tech_filter
Selecting qrc RC Corner tech (qrcTech) files.

read_libs (*library_types: typing.Iterable[hammer_vlsi_impl.LibraryFilter], output_func: typing.Callable[[str, hammer_vlsi_impl.LibraryFilter], typing.List[str]], pre_filters: typing.Iterable[typing.Callable[abc.Library, bool]] = [], must_exist: bool = True*) → typing.List[str]
Read the given libraries and return a list of strings according to some output format.

Parameters

- **library_types** – List of libraries to filter, specified as a list of LibraryFilter elements.
- **output_func** – Function which processes the outputs, taking in the filtered lib and the library filter which generated it.
- **must_exist** – Must each library item actually exist? Default: True (yes, they must exist)

Returns List of filtered libraries processed according output_func.

static replace_tcl_set (*variable: str, value: str, tcl_path: str, quotes: bool = True*) → None
Utility function to replaces a "set VARIABLE ..." line with set VARIABLE "value" in the given TCL script file.

Parameters

- **variable** – Variable name to replace
- **value** – Value to replace it with (default quoted)
- **tcl_path** – Path to the TCL script.
- **quotes** – (optional) Set to False to disable quoting of the value.

run (*hook_actions: typing.List[hammer_vlsi_impl.HammerToolHookAction] = []*) → bool
Run this tool.

Perform some setup operations to set up the config and tool environment, runs the tool-specific actions defined in steps, and collects the outputs.

Returns True if the tool finished successfully; false otherwise.

run_dir
Get the location of the run dir, a writable temporary information for use by the tool. This should return an absolute path.

Returns Path to the location of the library.

run_executable (*args: typing.List[str], cwd: str = None*) → str
Run an executable and log the command to the log while also capturing the output.

Parameters

- **args** – Command-line to run; each item in the list is one token. The first token should be the command to run.
- **cwd** – Working directory (leave as None to use the current working directory).

Returns Output from the command or an error message.

run_steps (*steps: typing.List[hammer_vlsi_impl.HammerToolStep], hook_actions: typing.List[hammer_vlsi_impl.HammerToolHookAction] = []*) → bool
Run the given steps, checking for errors/conditions between each step.

Parameters

- **steps** – List of steps.
- **hook_actions** – List of hook actions.

Returns Returns true if all the steps are successful.

set_database (*database: hammer_config.config_src.HammerDatabase*) → None
Set the settings database for use by the tool.

set_setting (*key: str, value: typing.Any*) → None
Set a runtime setting in the database.

steps
List of steps defined for the execution of this tool.

static tcl_append (*cmd: str, output_buffer: typing.List[str]*) → None
Helper function to echo and run a command.

Parameters

- **cmd** – TCL command to run
- **output_buffer** – Buffer in which to enqueue the resulting TCL lines.

technology
Get the technology library currently in use.

Returns HammerTechnology instance

timing_db_filter
Selecting Synopsys timing libraries (.db). Prefers CCS if available; picks NLDM as a fallback.

tlu_max_cap_filter
TLU+ max cap filter.

tlu_min_cap_filter
TLU+ min cap filter.

static to_command_line_args(*lib_item: str*, *filt: hammer_vlsi_impl.LibraryFilter*) → typing.List[str]
Generate command-line args in the form –<filt.tag> <lib_item>.

static to_plain_item(*lib_item: str*, *filt: hammer_vlsi_impl.LibraryFilter*) → typing.List[str]
Generate plain outputs in the form of <lib_item1> <lib_item2> ...

tool_dir
Get the location of the tool library.

Returns Path to the location of the library.

static verbose_tcl_append(*cmd: str, output_buffer: typing.List[str]*) → None
Helper function to echo and run a command.

Parameters

- **cmd** – TCL command to run
- **output_buffer** – Buffer in which to enqueue the resulting TCL lines.

verilog_synth_filter
Selecting verilog_synth files which are synthesizable wrappers (e.g. for SRAM) which are needed in some technologies.

class hammer_vlsi.**HammerToolHookAction**(*location*, *target_name*, *step*)
Bases: tuple

location
Alias for field number 0

step
Alias for field number 2

target_name
Alias for field number 1

exception hammer_vlsi.**HammerToolPauseException**
Bases: Exception

Internal hammer-vlsi exception raised to indicate that a step has stopped execution of the tool. This is not necessarily an error condition.

class hammer_vlsi.**HammerToolStep**(*func*, *name*)
Bases: tuple

func
Alias for field number 0

name
Alias for field number 1

class hammer_vlsi.**HammerVLSIFileLogger**(*output_path: str*, *format_msg_callback: typing.Callable[hammer_vlsi_impl.FullMessage, str]* = *None*) → None
Bases: object

A file logger for HammerVLSILogging.

callback
Get the callback for HammerVLSILogging.add_callback.

close() → None
Close this file logger.

class hammer_vlsi.**HammerVLSILogging**
Bases: object

Singleton which handles logging in hammer-vlsi.

This class is generally not intended to be used directly for logging, but through HammerVLSILoggingContext instead.

COLOUR_BLUE = '\x1b[96m'

COLOUR_CLEAR = '\x1b[0m'

COLOUR_GREY = '\x1b[37m'

COLOUR_RED = '\x1b[91m'

COLOUR_RED_BG = '\x1b[101m'

COLOUR_YELLOW = '\x1b[33m'

classmethod add_callback(*callback: typing.Callable[hammer_vlsi_impl.FullMessage, NoneType]*) → None
Add a callback.

classmethod build_log_message(*fullmessage: hammer_vlsi_impl.FullMessage*) → str
Build a plain message for logs, without colour.

classmethod build_message(*fullmessage: hammer_vlsi_impl.FullMessage*) → str
Build a colour message.

classmethod callback_buffering(*fullmessage: hammer_vlsi_impl.FullMessage*) → None
Get the current contents of the logging buffer and clear it.

classmethod callback_print(*fullmessage: hammer_vlsi_impl.FullMessage*) → None
Default callback which prints a colour message.

callbacks = [<bound method HammerVLSILogging.callback_print of <class 'hammer_vlsi_impl.HammerVLSILogging'>

classmethod clear_callbacks() → None
Clear the list of callbacks.

classmethod context(*new_context: str = ''*) → hammer_vlsi_impl.HammerVLSILoggingContext
Create a new context.

Parameters **new_context** – Context name. Leave blank to get the global context.

enable_buffering = False

enable_colour = True

enable_tag = True

classmethod get_buffer() → typing.Iterable[str]
Get the current contents of the logging buffer and clear it.

classmethod get_colour_escape(*level: hammer_vlsi_impl.Level*) → str
Colour table to translate level -> colour in logging.

static get_tag (*context: typing.List[str]*) → str
Helper function to get the tag for outputing a message given a context.

classmethod log (*fullmessage: hammer_vlsi_impl.FullMessage*) → None
Log the given message at the given level in the given context.

output_buffer = []

class hammer_vlsi.**HammerVLSILoggingContext** (*context: typing.List[str], logging_class: typing.Type[hammer_vlsi_impl.HammerVLSILogging]*) → None
Bases: object

Logging interface to hammer-vlsi which contains a context (list of strings denoting hierarchy where the log occurred). e.g. [”synthesis”, “subprocess run-synthesis”]

context (*new_context: str*) → VT
Create a new subcontext from this context.

debug (*message: str*) → None
Create an debug-level log message.

error (*message: str*) → None
Create an error-level log message.

fatal (*message: str*) → None
Create an fatal-level log message.

info (*message: str*) → None
Create an info-level log message.

log (*message: str*, *level: hammer_vlsi_impl.Level*) → None

warning (*message: str*) → None
Create an warning-level log message.

class hammer_vlsi.**HammerVLSISettings**
Bases: object

Static class which holds global hammer-vlsi settings.

static get_config () → dict
Export settings as a config dictionary.

hammer_vlsi_path = ‘’

classmethod set_hammer_vlsi_path_from_environment () → bool
Try to set hammer_vlsi_path from the environment variable HAMMER_VLSI.

Returns True if successfully set, False otherwise

class hammer_vlsi.**HasSDCSupport**
Bases: hammer_vlsi_impl.HammerTool

Mix-in trait with functions useful for tools with SDC-style constraints.

sdc_clock_constraints
Generate TCL fragments for top module clock constraints.

sdc_pin_constraints
Generate a fragment for I/O pin constraints.

class hammer_vlsi.**HierarchicalMode**
Bases: enum.Enum

An enumeration.

Flat = 1

Hierarchical = 3

Leaf = 2

Top = 4

static from_str(*x: str*) → hammer_vlsi_impl.HierarchicalMode

is_nonleaf_hierarchical() → bool
Helper function that returns True if this mode is a non-leaf hierarchical mode (i.e. any block with hierarchical sub-blocks).

class hammer_vlsi.**HookLocation**
Bases: enum.Enum

An enumeration.

InsertPostStep = 2

InsertPreStep = 1

ReplaceStep = 10

ResumePostStep = 21

ResumePreStep = 20

class hammer_vlsi.**ILMStruct**
Bases: hammer_vlsi_impl.ILMStruct

static from_setting(*ilm: dict*) → hammer_vlsi_impl.ILMStruct

to_setting() → dict

class hammer_vlsi.**Level**
Bases: enum.Enum

Logging levels.

DEBUG = 0

ERROR = 3

FATAL = 4

INFO = 1

WARNING = 2

class hammer_vlsi.**LibraryFilter**
Bases: hammer_vlsi_impl.LibraryFilter

static new(*tag: str, description: str, is_file: bool, extraction_func: typing.Callable[abc.Library, typing.List[str]], filter_func: typing.Union[typing.Callable[[abc.Library], bool], NoneType] = None, sort_func: typing.Union[typing.Callable[[abc.Library], typing.Union[numbers.Number, str, tuple]], NoneType] = None, extra_post_filter_funcs: typing.List[typing.Callable[typing.List[str], typing.List[str]]] = []*) → hammer_vlsi_impl.LibraryFilter
Convenience "constructor" with some default arguments.

class hammer_vlsi.**MMMCCorner**(*name*, *type*, *voltage*, *temp*)
Bases: tuple

name
Alias for field number 0

temp
Alias for field number 3

type
Alias for field number 1

voltage
Alias for field number 2

class hammer_vlsi.**MMMCCornerType**
Bases: enum.Enum

An enumeration.

Extra = 3

Hold = 2

Setup = 1

static from_string(*s: str*) → hammer_vlsi_impl.MMMCCornerType

class hammer_vlsi.**Margins**(*left*, *bottom*, *right*, *top*)
Bases: tuple

bottom
Alias for field number 1

left
Alias for field number 0

right
Alias for field number 2

top
Alias for field number 3

class hammer_vlsi.**OutputLoadConstraint**(*name*, *load*)
Bases: tuple

load
Alias for field number 1

name
Alias for field number 0

class hammer_vlsi.**PlacementConstraint**
Bases: hammer_vlsi_impl.PlacementConstraint

static from_dict(*constraint: dict*) → hammer_vlsi_impl.PlacementConstraint

to_dict() → dict

class hammer_vlsi.**PlacementConstraintType**
Bases: enum.Enum

An enumeration.

Dummy = 1

HardMacro = 4

Hierarchical = 5

Placement = 2

TopLevel = 3

static from_str (*x: str*) → hammer_vlsi_impl.PlacementConstraintType

class hammer_vlsi.**SynopsysTool**
Bases: hammer_vlsi_impl.HasSDCSupport, hammer_vlsi_impl.HammerTool

Mix-in trait with functions useful for Synopsys-based tools.

env_vars
Get the list of environment variables required for this tool. Note to subclasses: remember to include variables from super().env_vars!

get_synopsys_rm_tarball (*product: str*, *settings_key: str = ''*) → str
Locate reference methodology tarball.

Parameters

- **product** – Either "DC" or "ICC"
- **settings_key** – Key to retrieve the version for the product. Leave blank for DC and ICC.

class hammer_vlsi.**TimeValue** (*value: str*, *default_prefix: str = 'n'*) → None
Bases: object

Time value - e.g. "4 ns". Parses time values from strings.

str_value_in_units (*prefix: str*, *round_zeroes: bool = True*) → str
Get this time value in the given prefix but including the units. e.g. return "5 ns".

Parameters

- **prefix** – Prefix for the resulting value - e.g. "ns".
- **round_zeroes** – True to round 1.00000001 etc to 1 within 2 decimal places.

value
Get the value of this time value.

value_in_units (*prefix: str*, *round_zeroes: bool = True*) → float
Get this time value in the given prefix. e.g. "ns"

class hammer_vlsi.**VerilogUtils**
Bases: object

static contains_module (*v: str*, *module: str*) → bool
Check if the given Verilog source contains the given module.

Parameters

- **v** – Verilog source code
- **module** – Module to look for

Returns True if the given module exists.

static remove_comments (*v: str*) → str
Remove comments from the given Verilog file.

Parameters **v** – Verilog source code

Returns Source code without comments

static remove_module (*v: str*, *module: str*) → str
Remove the given module from the given Verilog source file, if it exists.

Parameters

- **v** – Verilog source code
- **module** – Module to remove

Returns Verilog with given module definition removed, if it exists

hammer_vlsi.**add_dicts**(*a: dict*, *b: dict*) → dict
Helper method: join two dicts together while type checking. The second dictionary will override any entries in the first.

hammer_vlsi.**add_lists**(*a: typing.List[str]*, *b: typing.List[str]*) → typing.List[str]
Helper method: join two lists together while type checking.

hammer_vlsi.**check_hammer_step_function**(*func: typing.Callable[_ForwardRef('HammerTool'), bool]*) → None

hammer_vlsi.**deepdict**(*x: dict*) → dict
Deep copy a dictionary. This is needed because dict() by itself only makes a shallow copy. See https://stackoverflow.com/questions/5105517/deep-copy-of-a-dict-in-python Convenience function.

Parameters **x** – Dictionary to copy

Returns Deep copy of the dictionary provided by copy.deepcopy().

hammer_vlsi.**deeplist**(*x: list*) → list
Deep copy a list. This is needed because list() by itself only makes a shallow copy. See https://stackoverflow.com/questions/5105517/deep-copy-of-a-dict-in-python Convenience function.

Parameters **x** – List to copy

Returns Deep copy of the list provided by copy.deepcopy().

hammer_vlsi.**in_place_unique**(*items: typing.List[typing.Any]*) → None
"Fast" in-place uniquification of a list.

Parameters **items** – List to be uniquified.

hammer_vlsi.**load_tool**(*tool_name: str*, *path: typing.Iterable[str]*) → hammer_vlsi_impl.HammerTool
Load the given tool. See the hammer-vlsi README for how it works.

Parameters

- **tool_name** – Name of the tool
- **path** – List of paths to get

Returns HammerTool of the given tool

hammer_vlsi.**make_raw_hammer_tool_step**(*func: typing.Callable[_ForwardRef('HammerTool'), bool]*, *name: str*) → hammer_vlsi_impl.HammerToolStep

hammer_vlsi.**reduce**(*function*, *sequence*[, *initial*]) → value
Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

hammer_vlsi.**reduce_named**(*function: typing.Callable*, *sequence: typing.Iterable*, *initial=None*) → typing.Any
Version of functools.reduce with named arguments. See https://mail.python.org/pipermail/python-ideas/2014-October/029803.html

hammer_vlsi.**reverse_dict**(*x: dict*) → dict

Reverse a dictionary (keys become values and vice-versa). Only works if the dictionary is isomorphic (no duplicate values), or some pairs will be lost.

Parameters **x** – Dictionary to reverse

Returns Reversed dictionary

hammer_vlsi.**topological_sort**(*graph: typing.Dict[str, typing.Tuple[typing.List[str], typing.List[str]]], starting_nodes: typing.List[str]*) → typing.List[str]

Perform a topological sort on the graph and return a valid ordering.

Parameters

- **graph** – dict that represents key as the node and value as a tuple of (outgoing edges, incoming edges).
- **starting_nodes** – List of starting nodes to use.

Returns A valid topological ordering of the graph.

hammer_vlsi.**with_default_callbacks**(*cls*)

C Hammer IR/Configuration Library Documentation

CHAPTER

TWO

HAMMER_CONFIG MODULE

class hammer_config.**HammerDatabase** → None
Bases: object

Define a database which is composed of a set of overridable configs. We need something like this in order to e.g. bind technology afterwards, since we never want technology to override project. If we just did an .update() with the technology config, we'd possibly lose the previously-bound project config.

Terminology: - setting: a single key-value pair e.g. "vlsi.core.technology" -> "footech" - config: a single concrete dictionary of settings. - database: a collection of configs with a specific override hierarchy.

Order of precedence (in increasing order): - builtins - core - tools - technology - environment - project - runtime (settings dynamically updated during the run a hammer run)

get(*key: str*) → typing.Any
Alias for get_setting().

get_config() → dict
Get the config of this database after all the overrides have been dealt with.

get_database_json() → str
Get the database (get_config) in JSON form as a string.

get_setting(*key: str*, *nullvalue: str = 'null'*) → typing.Any
Retrieve the given key.

Parameters

- **key** – Desired key.
- **nullvalue** – Value to return out for nulls.

Returns The given config

has_setting(*key: str*) → bool
Check if the given key exists in the database.

Parameters **key** – Desired key.

Returns True if the given setting exists.

static internal_keys() → typing.Set[str]
Internal keys that shouldn't show up in any final config.

runtime

set_setting(*key: str*, *value: typing.Any*) → None
Set the given key. The setting will be placed into the runtime dictionary.

Parameters

- **key** – Key
- **value** – Value for key

update_builtins (*builtins_config: typing.List[dict]*) → None
Update the builtins config with the given builtins config.

update_core (*core_config: typing.List[dict]*) → None
Update the core config with the given core config.

update_environment (*environment_config: typing.List[dict]*) → None
Update the environment config with the given environment config.

update_project (*project_config: typing.List[dict]*) → None
Update the project config with the given project config.

update_technology (*technology_config: typing.List[dict]*) → None
Update the technology config with the given technology config.

update_tools (*tools_config: typing.List[dict]*) → None
Update the tools config with the given tools config.

hammer_config.**combine_configs** (*configs: typing.Iterable[dict]*) → dict
Combine the given list of *unpacked* configs into a single config. Later configs in the list will override the earlier configs.

Parameters

- **configs** – List of configs.
- **handle_meta** – Handle meta configs?

Returns A loaded config dictionary.

hammer_config.**deepdict** (*x: dict*) → dict
Deep copy a dictionary. This is needed because dict() by itself only makes a shallow copy. See https://stackoverflow.com/questions/5105517/deep-copy-of-a-dict-in-python Convenience function.

Parameters **x** – Dictionary to copy

Returns Deep copy of the dictionary provided by copy.deepcopy().

hammer_config.**load_config_from_defaults** (*path: str*, *strict: bool = False*) → typing.List[dict]
Load the default configuration for a hammer-vlsi tool/library/technology in the given path, which consists of defaults.yml and defaults.json (with defaults.json taking priority).

Parameters

- **config_paths** – Path to defaults.yml and defaults.json.
- **strict** – Set to true to error if the file is not found.

Returns A list of configs in increasing order of precedence.

hammer_config.**load_config_from_file** (*filename: str*, *strict: bool = False*) → dict
Load config from a filename, returning a blank dictionary if the file is empty, instead of an error. Supports .yml and .json, and will raise an error otherwise.

Parameters

- **filename** – Filename to the config in .yml or .json.
- **strict** – Set to true to error if the file is not found.

Returns Loaded config dictionary, unpacked.

hammer_config.**load_config_from_paths**(*config_paths: typing.Iterable[str], strict: bool = False*) → typing.List[dict]

Load configuration from paths containing *.yml and *.json files. As noted in README.config, .json will take precedence over .yml files.

Parameters

- **config_paths** – Path to *.yml and *.json config files.
- **strict** – Set to true to error if the file is not found.

Returns A list of configs in increasing order of precedence.

hammer_config.**load_config_from_string**(*contents: str*, *is_yaml: bool*, *path: str = 'unspecified'*) → dict

Load config from a string by loading it and unpacking it.

Parameters

- **contents** – Contents of the config.
- **is_yaml** – True if the contents are yaml.
- **path** – Path to the folder where the config file is located.

Returns Loaded config dictionary, unpacked.

hammer_config.**load_yaml**(*yamlStr: str*) → dict

Load a YAML database as JSON.

The input file is parsed as YAML and converted to a python dict tree, then that tree is converted to the JSON output. There is a check to make sure the two dict trees are structurally identical.

Parameters **yamlStr** – A string containing the yaml database.

Returns A dictionary object representing the yaml database.

hammer_config.**reduce**(*function*, *sequence*[, *initial*]) → value

Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

hammer_config.**unpack**(*config_dict: dict*, *prefix: str = ''*) → dict

Unpack the given config_dict, flattening key names recursively. >>> p = unpack({"one": 1, "two": 2}, prefix="snack") >>> p == {'snack.one': 1, 'snack.two': 2} True >>> p = unpack({"a": {"foo": 1, "bar": 2}}) >>> p == {'a.foo': 1, 'a.bar': 2} True >>> p = unpack({"a.b": {"foo": 1, "bar": 2}}) >>> p == {"a.b.foo": 1, "a.b.bar": 2} True >>> p = unpack({ ... "a": { ... "foo": 1, ... "bar": 2 ... }, ... "b": { ... "baz": 3, ... "boom": {"rocket": "chip", "hwacha": "vector"} ... }, ... }) >>> p == {"a.foo": 1, "a.bar": 2, "b.baz": 3, "b.boom.rocket": "chip", ... "b.boom.hwacha": "vector"} True

hammer_config.**update_and_expand_meta**(*config_dict: dict*, *meta_dict: dict*) → dict

Expand the meta directives for the given config dict and return a new dictionary containing the updated settings with respect to the base config_dict.

Parameters

- **config_dict** – Base config.
- **meta_dict** – Dictionary with potentially new meta directives.

Returns New dictionary with meta_dict updating config_dict.

D Hammer Technology API Documentation

CHAPTER

THREE

HAMMER_TECH MODULE

class hammer_tech.**HammerTechnology**

Bases: object

cache_dir

Get the location of a cache dir for this library.

Returns Path to the location of the cache dir.

check_installs() → bool

Check that the all directories for a pre-installed technology actually exist.

Returns Return True if the directories is OK, False otherwise.

extract_tarballs() → None

Extract tarballs to the given cache_dir, or verify that they've been extracted.

extract_technology_files() → None

Ensure that the technology files exist either via tarballs or installs.

extracted_tarballs_dir

Return the path to a folder under self.path where extracted tarballs are stored/cached.

get_config() → typing.List[dict]

Get the hammer configuration for this technology. Not to be confused with the ".tech.json" which self.config refers to.

get_setting(*key: str*)

Get a particular setting from the database.

classmethod load_from_dir(*technology_name: str*, *path: str*)

Load a technology from a given folder.

Parameters

- **technology_name** – Technology name (e.g. "saed32")
- **path** – Path to the technology folder (e.g. foo/bar/technology/saed32)

logger

Get the logger for this tool.

prepend_dir_path(*path: str*) → str

Prepend the appropriate path (either from tarballs or installs) to the given library item.

set_database(*database: hammer_config.config_src.HammerDatabase*) → None

Set the settings database for use by the tool.

class hammer_tech.**HammerVLSILoggingContext**(*context: typing.List[str], logging_class: typing.Type[hammer_logging.HammerVLSILogging]*) → None

Bases: object

Logging interface to hammer-vlsi which contains a context (list of strings denoting hierarchy where the log occurred). e.g. [”synthesis”, “subprocess run-synthesis”]

context(*new_context: str*) → hammer_logging.HammerVLSILoggingContext
Create a new subcontext from this context.

debug(*message: str*) → None
Create an debug-level log message.

error(*message: str*) → None
Create an error-level log message.

fatal(*message: str*) → None
Create an fatal-level log message.

info(*message: str*) → None
Create an info-level log message.

log(*message: str*, *level: hammer_logging.Level*) → None

warning(*message: str*) → None
Create an warning-level log message.

class hammer_tech.**Library**(***props*)

Bases: python_jsonschema_objects.classbuilder.ProtocolBase

ccs_liberty_file

ccs_library_file

corner

itf_files

lef_file

metal_layers

milkyway_lib_in_dir

milkyway_techfile

nldm_liberty_file

nldm_library_file

openaccess_techfile

provides

qrc_techfile

supplies

tluplus_files

verilog_sim

verilog_synth

hammer_tech.**TechJSON**
alias of techjson

E Floorplanning DSL documentation

This appendix section contains documentation for the key data structures/types in the floorplanning DSL. For full documentation, please see the appropriate source repository.

floorplan

LayoutBase

Related Docs: object LayoutBase | package floorplan

trait LayoutBase extends AnyRef

Base type for all layout elements. NOTE: all layout engine data structures are immutable. replace*() methods will create a new copy of the data structure with the given field altered.

Linear Supertypes
AnyRef, Any
Known Subclasses
ArrayLayout, ArrayLayoutImpl, AutoLayout, Expander, ExpanderImpl, HBox, HardMacro, HorizontalExpander, LayoutBaseImpl, VBox, VerticalExpander

Ordering

1. Alphabetic
2. By Inheritance

Inherited

1. LayoutBase
2. AnyRef
3. Any

1. Hide All
2. Show All

Visibility

1. Public
2. All

Abstract Value Members

1. **abstract def create(newProperties: PropertyMap): LayoutBase.this.type**

   Create a copy of this object but replacing the properties with the given ones.

   Create a copy of this object but replacing the properties with the given ones.

newProperties

New properties to put in the object.

returns

A new copy of this object with replaced properties.

2. **abstract def properties: PropertyMap**

Concrete Value Members

1. **def allConstraints: LayoutConstraints**

Get all constraints for this block, recursively, named relative to this layout block.

Get all constraints for this block, recursively, named relative to this layout block. It will append names hierarchically to solve.

returns

List of all constraints (recursive)

2. **def dimensions: Dimensions**

3. **def get[T](key: PropertyKey, other: Option[T] = None): T**

4. **def hasFixedDimensions: Boolean**

Determines if the dimensions for this block are fixed.

Determines if the dimensions for this block are fixed. For example, this is true for hard macros or a box whose elements are all hard macros.

returns

true if the dimensions for this block are fixed.

5. **def height: Option[LayoutDimension]**

6. **def heightConstraintName: String**

7. **def hierarchicalAddName(constraints: LayoutConstraints): LayoutConstraints**

8. **def localConstraints: LayoutConstraints**

Get all local constraints (non-recursive), named relative to this layout block.

Get all local constraints (non-recursive), named relative to this layout block. e.g. if this is a HBox named "a" with two elements x and y, it will return a list of "a=x+y", "x=a-y", "y=a-x".

returns

List of local constraints

9. **def localConstraintsInternal: LayoutConstraints**

10. **def name: String**

11. **def path: Option[Path]**

12. **def replaceCoordinates(x: LayoutDimension, y: LayoutDimension): LayoutBase.this.type**

13. **def replaceDimensions(dimensions: Dimensions): LayoutBase.this.type**

14. **def replaceHeight(height: LayoutDimension): LayoutBase.this.type**

15. **def replaceName(name: String): LayoutBase.this.type**

16. **def replacePath(path: Option[Path]): LayoutBase.this.type**

17. **def replacePath(path: Path): LayoutBase.this.type**

18. **def replacePathViaCommand(command: Command)(implicit cc: CircuitContext, options: CircuitContextOptions): LayoutBase.this.type**

Attach the given FCL selector to this item via CircuitContext.selectOne().

Attach the given FCL selector to this item via CircuitContext.selectOne().

command

Command to select.

cc

CircuitContext to use.

options

CircuitContext options.

19. **def replaceProperties(props: Seq[(PropertyKey, AnyRef)]): LayoutBase.this.type**

20. **def replaceProperty(key: PropertyKey, value: AnyRef): LayoutBase.this.type**

21. **def replaceWidth(width: LayoutDimension): LayoutBase.this.type**

22. **def replaceWidthAndHeight(width: LayoutDimension, height: LayoutDimension): LayoutBase.this.type**

23. **def resolve(updateCoordinates: Boolean = true)(implicit verbosity: Verbosity = ...): LayoutBase.this.type**

Attempt to fully resolve all widths and heights in this layout using the constraint engine.

Attempt to fully resolve all widths and heights in this layout using the constraint engine.

updateCoordinates

Update co-ordinates? (true by default)

returns

A fully resolved version of this layout block.

24. **def width: Option[LayoutDimension]**

25. **def widthConstraintName: String**

26. **def x: Option[LayoutDimension]**

27. **def y: Option[LayoutDimension]**

floorplan

Dimensions

Related Doc: package floorplan

case class Dimensions(x: Option[LayoutDimension], y: Option[LayoutDimension], width: Option[LayoutDimension], height: Option[LayoutDimension]) extends Product with Serializable

Rectangular dimensions to specify the location and size. Note: the dimensions are cartesian - increasing x moves to the right, increasing y moves up.

x

x-coordinate

y

y-coordinate

width

Width of the object

height

Height of the object

Linear Supertypes
Serializable, Serializable, Product, Equals, AnyRef, Any

Ordering

1. Alphabetic
2. By Inheritance

Inherited

1. Dimensions
2. Serializable
3. Serializable
4. Product
5. Equals
6. AnyRef
7. Any

1. Hide All
2. Show All

Visibility

1. Public
2. All

Instance Constructors

1. **new Dimensions(x: Option[LayoutDimension], y: Option[LayoutDimension], width: Option[LayoutDimension], height: Option[LayoutDimension])**

   x

   x-coordinate

   y

   y-coordinate

   width

   Width of the object

   height

   Height of the object

Value Members

1. **val height: Option[LayoutDimension]**

   Height of the object

2. **val width: Option[LayoutDimension]**

   Width of the object

3. **val x: Option[LayoutDimension]**

   x-coordinate

4. **val y: Option[LayoutDimension]**

y-coordinate

floorplan

ArrayLayout

Related Docs: object ArrayLayout | package floorplan

trait ArrayLayout extends LayoutBase

Linear Supertypes
LayoutBase, AnyRef, Any
Known Subclasses
ArrayLayoutImpl, HBox, VBox

Ordering

1. Alphabetic
2. By Inheritance

Inherited

1. ArrayLayout
2. LayoutBase
3. AnyRef
4. Any

1. Hide All
2. Show All

Visibility

1. Public
2. All

Abstract Value Members

1. **abstract def create(newProperties: PropertyMap): ArrayLayout.this.type**

   Create a copy of this object but replacing the properties with the given ones.

   Create a copy of this object but replacing the properties with the given ones.

   newProperties

New properties to put in the object.

returns

A new copy of this object with replaced properties.

Definition Classes
LayoutBase

2. **abstract def elements: Seq[LayoutBase]**

3. **abstract def properties: PropertyMap**

Definition Classes
LayoutBase

4. **abstract def replaceElements(elements: Seq[LayoutBase]): ArrayLayout.this.type**

Concrete Value Members

1. **def allConstraints: LayoutConstraints**

Get all constraints for this block, recursively, named relative to this layout block.

Get all constraints for this block, recursively, named relative to this layout block. It will append names hierarchically to solve.

returns

List of all constraints (recursive)

Definition Classes
LayoutBase

2. **def dimensions: Dimensions**

Definition Classes
LayoutBase

3. **def get[T](key: PropertyKey, other: Option[T] = None): T**

Definition Classes
LayoutBase

4. **def hasFixedDimensions: Boolean**

Determines if the dimensions for this block are fixed.

Determines if the dimensions for this block are fixed. For example, this is true for hard macros or a box whose elements are all hard macros.

returns

true if the dimensions for this block are fixed.

Definition Classes
LayoutBase

5. **def height: Option[LayoutDimension]**

Definition Classes
LayoutBase

6. **def heightConstraintName: String**

Definition Classes
LayoutBase

7. **def hierarchicalAddName(constraints: LayoutConstraints): LayoutConstraints**

Definition Classes
LayoutBase

8. **def localConstraints: LayoutConstraints**

Get all local constraints (non-recursive), named relative to this layout block.

Get all local constraints (non-recursive), named relative to this layout block. e.g. if this is a HBox named "a" with two elements x and y, it will return a list of "a=x+y", "x=a-y", "y=a-x".

returns

List of local constraints

Definition Classes
LayoutBase

9. **def localConstraintsInternal: LayoutConstraints**

Definition Classes
LayoutBase

10. **def name: String**

Definition Classes
LayoutBase

11. **def path: Option[Path]**

Definition Classes
LayoutBase

12. **def replaceCoordinates(x: LayoutDimension, y: LayoutDimension): ArrayLayout.this.type**

Definition Classes
LayoutBase

13. **def replaceDimensions(dimensions: Dimensions): ArrayLayout.this.type**

Definition Classes
LayoutBase

14. **def replaceHeight(height: LayoutDimension): ArrayLayout.this.type**

Definition Classes
LayoutBase

15. **def replaceName(name: String): ArrayLayout.this.type**

Definition Classes
LayoutBase

16. **def replacePath(path: Option[Path]): ArrayLayout.this.type**

Definition Classes
LayoutBase

17. **def replacePath(path: Path): ArrayLayout.this.type**

Definition Classes
LayoutBase

18. **def replacePathViaCommand(command: Command)(implicit cc: CircuitContext, options: CircuitContextOptions): ArrayLayout.this.type**

Attach the given FCL selector to this item via CircuitContext.selectOne().

Attach the given FCL selector to this item via CircuitContext.selectOne().

command

Command to select.

cc

CircuitContext to use.

options

CircuitContext options.

Definition Classes
LayoutBase

19. **def replaceProperties(props: Seq[(PropertyKey, AnyRef)]): ArrayLayout.this.type**

Definition Classes
LayoutBase

20. **def replaceProperty(key: PropertyKey, value: AnyRef): ArrayLayout.this.type**

Definition Classes
LayoutBase

21. **def replaceWidth(width: LayoutDimension): ArrayLayout.this.type**

Definition Classes
LayoutBase

22. **def replaceWidthAndHeight(width: LayoutDimension, height: LayoutDimension): ArrayLayout.this.type**

Definition Classes
LayoutBase

23. **def resolve(updateCoordinates: Boolean = true)(implicit verbosity: Verbosity = ...): ArrayLayout.this.type**

Attempt to fully resolve all widths and heights in this layout using the constraint engine.

Attempt to fully resolve all widths and heights in this layout using the constraint engine.

updateCoordinates

Update co-ordinates? (true by default)

returns

A fully resolved version of this layout block.

Definition Classes
LayoutBase

24. **def unsizedSumConstraintName: String**

Special constraint for the sum of all unsized expanders.

Special constraint for the sum of all unsized expanders. Used when multiple expanders exist in a box and need to be resolved.

25. **def width: Option[LayoutDimension]**

Definition Classes
LayoutBase

26. **def widthConstraintName: String**

Definition Classes
LayoutBase

27. **def x: Option[LayoutDimension]**

Definition Classes
LayoutBase

28. **def y: Option[LayoutDimension]**

Definition Classes
LayoutBase

floorplan

HardMacro

Related Docs: object HardMacro | package floorplan

class HardMacro extends LayoutBaseImpl with LayoutBase

Hard macro block. Basically a big black box with fixed dimensions which needs to be placed somewhere.

Linear Supertypes
LayoutBaseImpl, LayoutBase, AnyRef, Any

Ordering

1. Alphabetic
2. By Inheritance

Inherited

1. HardMacro
2. LayoutBaseImpl
3. LayoutBase
4. AnyRef
5. Any

1. Hide All
2. Show All

Visibility

1. Public
2. All

Instance Constructors

1. **new HardMacro(properties: PropertyMap)**

Value Members

1. **def allConstraints: LayoutConstraints**

Get all constraints for this block, recursively, named relative to this layout block.

Get all constraints for this block, recursively, named relative to this layout block. It will append names hierarchically to solve.

returns

List of all constraints (recursive)

Definition Classes
LayoutBase

2. **def create(newProperties: PropertyMap): HardMacro.this.type**

Create a copy of this object but replacing the properties with the given ones.

Create a copy of this object but replacing the properties with the given ones.

newProperties

New properties to put in the object.

returns

A new copy of this object with replaced properties.

Definition Classes
HardMacro → LayoutBase

3. **def dimensions: Dimensions**

Definition Classes
LayoutBase

4. **def get[T](key: PropertyKey, other: Option[T] = None): T**

Definition Classes
LayoutBase

5. **def hasFixedDimensions: Boolean**

Determines if the dimensions for this block are fixed.

Determines if the dimensions for this block are fixed. For example, this is true for hard macros or a box whose elements are all hard macros.

returns

true if the dimensions for this block are fixed.

Definition Classes
HardMacro → LayoutBase

6. **def height: Option[LayoutDimension]**

Definition Classes
LayoutBase

7. **def heightConstraintName: String**

Definition Classes
LayoutBase

8. **def hierarchicalAddName(constraints: LayoutConstraints): LayoutConstraints**

Definition Classes
LayoutBase

9. **def localConstraints: LayoutConstraints**

Get all local constraints (non-recursive), named relative to this layout block.

Get all local constraints (non-recursive), named relative to this layout block. e.g. if this is a HBox named "a" with two elements x and y, it will return a list of "a=x+y", "x=a-y", "y=a-x".

returns

List of local constraints

Definition Classes
LayoutBase

10. **def localConstraintsInternal: LayoutConstraints**

Definition Classes

LayoutBase

11. **def name: String**

Definition Classes
LayoutBase

12. **def path: Option[Path]**

Definition Classes
LayoutBase

13. **val properties: PropertyMap**

Definition Classes
HardMacro → LayoutBaseImpl → LayoutBase

14. **def replaceCoordinates(x: LayoutDimension, y: LayoutDimension): HardMacro.this.type**

Definition Classes
LayoutBase

15. **def replaceDimensions(dimensions: Dimensions): HardMacro.this.type**

Definition Classes
LayoutBase

16. **def replaceHeight(height: LayoutDimension): HardMacro.this.type**

Definition Classes
LayoutBase

17. **def replaceName(name: String): HardMacro.this.type**

Definition Classes
LayoutBase

18. **def replacePath(path: Option[Path]): HardMacro.this.type**

    Definition Classes
    LayoutBase

19. **def replacePath(path: Path): HardMacro.this.type**

    Definition Classes
    LayoutBase

20. **def replacePathViaCommand(command: Command)(implicit cc: CircuitContext, options: CircuitContextOptions): HardMacro.this.type**

    Attach the given FCL selector to this item via CircuitContext.selectOne().

    Attach the given FCL selector to this item via CircuitContext.selectOne().

    command

    Command to select.

    cc

    CircuitContext to use.

    options

    CircuitContext options.

    Definition Classes
    LayoutBase

21. **def replaceProperties(props: Seq[(PropertyKey, AnyRef)]): HardMacro.this.type**

    Definition Classes
    LayoutBase

22. **def replaceProperty(key: PropertyKey, value: AnyRef): HardMacro.this.type**

    Definition Classes
    LayoutBase

23. **def replaceWidth(width: LayoutDimension): HardMacro.this.type**

    Definition Classes
        LayoutBase

24. **def replaceWidthAndHeight(width: LayoutDimension, height: LayoutDimension): HardMacro.this.type**

    Definition Classes
        LayoutBase

25. **def resolve(updateCoordinates: Boolean = true)(implicit verbosity: Verbosity = ...): HardMacro.this.type**

    Attempt to fully resolve all widths and heights in this layout using the constraint engine.

    Attempt to fully resolve all widths and heights in this layout using the constraint engine.

    updateCoordinates

        Update co-ordinates? (true by default)

    returns

        A fully resolved version of this layout block.

    Definition Classes
        LayoutBase

26. **def width: Option[LayoutDimension]**

    Definition Classes
        LayoutBase

27. **def widthConstraintName: String**

    Definition Classes
        LayoutBase

28. **def x: Option[LayoutDimension]**

Definition Classes
LayoutBase

29. **def y: Option[LayoutDimension]**

Definition Classes
LayoutBase

Glossary

place-and-route The process of physically actualizing a design by placing structural netlist of standard-cell gates and macros onto a given chip area and routing all connections between them, while obeying a certain set of constraints. 7–13, 15, 18, 22, 25, 30, 35, 90

process design kit A set of design files provided by a semiconductor foundry which includes (but is not limited to) technology description files (e.g. TLEFs, techfiles) which describe number of layers, layer functions, etc; standard cell libraries and associated layout, timing, and simulation information; IP blocks/memory macros and associated files. 6, 11, 90

synthesis The process of mapping a RTL description of a circuit into a structural netlist of standard-cell gates. 7, 11, 15, 18, 22, 24, 30, 90

system-on-chip A chip which contains not only a CPU/microcontroller but also including memory systems, peripherals, accelerators (e.g. GPUs, vector units, co-processors). For mixed-signal SoCs, this also includes analog/RF peripherals like on-chip radios, regulators, and sensing circuits. 10, 33, 90

tapein A tapein refers to the product of the exercise of producing a fabricatable chip design without actually fabricating it. This can help provide valuable feedback at the RTL and architectural level on the feasibility of a design without committing to the full cost of fabricating the chip. 9

tapeout Tapeout/taping out refers to the entire process of designing a chip to be manufactured by a semiconductor fab, including design, layout, VLSI (including synthesis and place-and-route), verification, and signoff checks (DRC/LVS), resulting in a GDS file to be sent to the bad. Named so because in pre-CAD days, IC designs would actually be physically defined using tape, hence the name to "tape out" a design. 5–7, 10

Acronyms

PDK process design kit 6, 11

SoC system-on-chip 10, 33

References

[1] SFGATE, "How many watts does an electric range pull?." http://homeguides.sfgate.com/many-watts-electric-range-pull-87939.html.

[2] PBS, "ENIAC: A pioneering computer." http://web.archive.org/web/20180502170912/http://www.pbs.org/transistor/science/events/eniac.html, 1999.

[3] United States War Department, "Physical aspects, operation of ENIAC are described." http://americanhistory.si.edu/comphist/pr4.pdf, 1946.

[4] Linustechtips forum users, "Transistor count in the 8700k." https://linustechtips.com/main/topic/897028-transistor-count-in-the-8700k/, 2018.

[5] Intel, "Intel Core i7-8700K desktop processor 6 cores up to 4.7GHz turbo unlocked lga1151 300 series 95w bx80684i78700k." http://web.archive.org/web/20180502172601/https://www.amazon.com/Intel-i7-8700K-Processor-Unlocked-BX80684i78700K/dp/B07598VZR8, 2018.

[6] Proclockers, "Intel Core i7-8700K CPU review." https://proclockers.com/reviews/cpus/intel-core-i7-8700k-cpu-review/page/0/3, 2018.

[7] M. M. Waldrop, "The chips are down for Moore's law," *Nature*, vol. 530, pp. 144–147, feb 2016.

[8] J. Larus, "Spending Moore's dividend," *Communications of the ACM*, vol. 52, p. 62, May 2009.

[9] J. Atwood, "Hardware is cheap, programmers are expensive." https://blog.codinghorror.com/hardware-is-cheap-programmers-are-expensive/, Dec. 2008.

[10] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFETs with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct 1974.

[11] K. Rupp, "42 years of microprocessor trend data." https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, Feb. 2018.

[12] P. Singer, "High cost per wafer, long design cycles may delay 20nm and beyond." http://electroiq.com/petes-posts/2014/01/22/high-cost-per-wafer-long-design-cycles-may-delay-20nm-and-beyond/, Jan. 2014.

[13] Quartz Corp, "The future of semiconductor: Scaling down node size." http://www.thequartzcorp.com/en/blog/2015/03/09/the-future-of-semiconductor-scaling-down-node-size/123, Mar. 2015.

[14] E. Sperling, "Is 7nm the last major node?." https://semiengineering.com/7nm-last-major-node/, July 2017.

[15] A. Olofsson, "Debunking the myth of the $100m ASIC," *EE Times*, Oct. 2011.

[16] E. Sperling, "How much will that chip cost?." http://semiengineering.com/how-much-will-that-chip-cost/, Mar. 2014.

[17] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: decoupling algorithms from schedules for high-performance image processing," *Communications of the ACM*, vol. 61, pp. 106–115, Dec. 2017.

[18] S. Sutherland, "Verilog HDL quick reference guide." http://sutherland-hdl.com/pdfs/verilog_2001_ref_guide.pdf, 2001.

[19] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

[20] M. Fowler, "Inversion of control." http://web.archive.org/web/20180503202813/https://martinfowler.com/bliki/InversionOfControl.html, June 2005.

[21] R. Sundaram, "Staying close to upstream projects." http://web.archive.org/web/20180503201623/https://fedoraproject.org/wiki/Staying_close_to_upstream_projects, Jan. 2016.

[22] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, *Chisel: constructing hardware in a Scala embedded language*, pp. 1216–1225. ACM, 2012.

[23] EPFL, "Frequently Asked Questions - Java interoperability." https://www.scala-lang.org/old/faq/4, 2012.

[24] D. Spiewak, "Interop between Java and Scala." http://www.codecommit.com/blog/java/interop-between-java-and-scala, Feb. 2009.

[25] Free Software Foundation, "How to Use Inline Assembly Language in C Code." https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gcc/Using-Assembly-Language-with-C.html, 2018.

[26] H. D. Benington, "Production of large computer programs," *IEEE Annals of the History of Computing*, vol. 5, pp. 350–361, Oct. 1983.

[27] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001.

[28] A. Fox and D. Patterson, "Using MOOCs to Reinvigorate Software Engineering Education." http://web.archive.org/web/20180504080758/https://pdfs.semanticscholar.org/presentation/ffc1/1fa48836f0a565bfbb1488ca196eeb434af8.pdf.

[29] Anysilicon, "ASIC design flow - an overview." https://anysilicon.com/asic-design-flow/, Aug. 2014.

[30] N. Horspool and P. Gorman, "Phases of an ASIC Project," in *The ASIC handbook*, Prentice Hall modern semiconductor design series, pp. 1–34, Upper Saddle River, NJ: Prentice Hall, 2001.

[31] J. Crossley, A. Puggelli, H.-P. Le, B. Yang, R. Nancollas, K. Jung, L. Kong, N. Narevsky, Y. Lu, N. Sutardja, E. J. An, A. L. Sangiovanni-Vincentelli, and E. Alon, "BAG: A designer-oriented integrated framework for the development of AMS circuit generators," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, (Piscataway, NJ, USA), pp. 74–81, IEEE Press, 2013.

[32] M. Reeves and M. Deimler, "Adaptability: The New Competitive Advantage," in *Own the Future* (M. Deimler, R. Lesser, D. Rhodes, and J. Sinha, eds.), pp. 19–26, Hoboken, NJ, USA: John Wiley & Sons, Inc., Aug. 2015.

[33] Value Investing World, "Warren Buffett on projections." http://www.valueinvestingworld.com/2013/05/warren-buffett-on-projections.html, May 2013.

[34] P.-F. Chiu, C. Celio, K. Asanović, D. Patterson, and B. Nilolic, "An out-of-order RISC-V processor with resilient low-voltage operation in 28nm CMOS," in *2018 Symposium on VLSI Circuits*, June 2018.

[35] S. Bailey, J. Wright, N. Mehta, R. Hochman, R. Jarnot, V. Milovanović, D. Werthimer, and B. Nikolić, "A 28nm FDSOI 8192-point digital ASIC spectrometer from a Chisel generator," in *CICC*, Apr. 2018.

[36] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," pp. 209–216, IEEE, Nov. 2017.

[37] K. Shahookar and P. Mazumder, "VLSI cell placement techniques," *ACM Computing Surveys*, vol. 23, pp. 143–220, June 1991.

[38] M. R. Cramer and J. V. Leeuwen, "Wire Routing is NP-Complete," Tech. Rep. RUU-CS-82-4, Department of Computer Science, University of Utrecht, Feb 1982.

[39] P. Armer, "SHARE-A Eulogy to Cooperative Effort," *IEEE Annals of the History of Computing*, vol. 2, pp. 122–129, Apr. 1980.

[40] E. W. Dijkstra, "Chapter I: Notes on structured programming," in *Structured Programming* (O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds.), pp. 1–82, London, UK, UK: Academic Press Ltd., 1972.

[41] L. Truong, D. Durst, B. Shacklett, O. Rizwan, and P. Hanrahan, "Magma." https://github.com/phanrahan/magma/tree/ea3517e06975980790163fe382c0ad8c707e2939, May 2018.

[42] I. Lewis, "Mixins and Python." https://www.ianlewis.org/en/mixins-and-python, Jan. 2013.