

Article

Characterizing and Comparing Direct-Tcl and Indirect-Abstract Approaches for Hardware Build Systems

Michał Kruszewski 

Institute of Electronic Systems, Faculty of Electronics and Information Technology,
Warsaw University of Technology, Nowowiejska 15/19, 00-650 Warsaw, Poland; michal.kruszewski@pw.edu.pl

Abstract

Build systems become an indispensable part of the software implementation and deployment process. New programming languages are released with the build system integrated into the language tools, for example, Go, Rust, or Zig. However, in the hardware description domain, no official build systems have been released with the predominant Hardware Description Languages (HDL) such as VHDL or SystemVerilog. Moreover, hardware design projects are often multilingual. This paper characterizes and compares two common approaches for hardware build system implementations. The first one is the direct-Tcl approach, in which the build system code is executed directly by the EDA tool during the design build flow. The second one is the indirect-abstract approach, in which the build system produces a Tcl script, which is later run by a proper EDA tool. As none of the existing direct-Tcl build systems were close to the indirect-abstract build systems in terms of supported functionalities, this paper also presents a new direct-Tcl hardware build system called HBS. The implemented build system was used as a representative of direct-Tcl build systems in comparison with indirect-abstract build systems. The main goals of the system include simplicity, readability, a minimal number of dependencies, ease of integration with the existing EDA tools, and independence from version control systems. The system provides a broad design automation, including support for multiple synthesis EDA tools, listing automatically detected cores and targets, automatic testbench targets detection, parallel testbench running, and dependency graph generation.

Keywords: automation; build system; electronic design automation; hardware synthesis; project maintenance; testbench runner



Academic Editors: Antonios N.
Dadaliaris and George I. Stamoulis

Received: 22 June 2026

Revised: 16 July 2026

Accepted: 24 July 2026

Published: 26 July 2026

Copyright: © 2026 by the author.
Licensee MDPI, Basel, Switzerland.
This article is an open access article
distributed under the terms and
conditions of the [Creative Commons
Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Implementation of any complex software or hardware description design involves more than code writing. Equally important is defining metadata used for configuration, dependency management, code generation, testing, and deployment purposes. The metadata is read by software tools, which, in turn, generate final binary or bitstream files or perform additional tasks, such as unit tests. Such programs are often referred to as build systems. Some build systems are split into multiple smaller programs, and some have a monolithic structure. An introduction to the build system concept and principles can be found in [1,2].

In the software domain, there are multiple well-established build systems, such as GNU Make [3], CMake [4], Ninja [5], or Bazel [6,7]. Some of them were implemented in the early days of programming. For example, GNU Make was first released in 1988 as part of the GNU Project. It was implemented to provide a free software alternative to Unix Make, which was first implemented in 1976 by Stuart Feldman at Bell Labs [8]. Modern

programming languages, such as Go, Rust, or Zig, include build systems in their toolchains. The Zig build system can even be used as a build system for C/C++ projects.

In the hardware domain, the situation looks different. The earliest Electronic Design Automation (EDA) tools were implemented as proprietary programs. EDA vendors did not (and still do not) cooperate to create a standard (or at least similar) user interface for tool interaction. The main common point among various EDA tools is the use of the Tcl language as the interpreter for interacting with the tools. However, the tools provide custom commands and parameters even for primary, common tasks, such as adding hardware description files to a project design. For example, to add a VHDL file in AMD Vivado, one can use specific `read_vhdl <path_to_vhdl_file>` or generic `add_files <path_to_vhdl_file>` commands. To add a VHDL file in Intel/Altera Quartus, one has to use the `set_global_assignment -name VHDL_FILE <path_to_vhdl_file>` command. EDA tools have an abundant amount of differences, making implementing a build system for hardware projects harder than for software. Despite this, there have been (and still are) attempts to implement build systems better suited to hardware projects.

The main goal of this paper is to characterize and compare two existing approaches for hardware build system implementation, the direct-Tcl and indirect-abstract. This paper also partially describes the new hardware build system (HBS), which follows the direct-Tcl approach. The system was designed and implemented to serve as a comparison representative.

The article's structure is as follows. Section 2 describes two commonly used approaches for implementing hardware build systems. Section 3.1 explains why HBS was implemented. Section 3.2 describes requirements imposed on the HBS implementation and user interface. Section 3.3 briefly introduces the internal architecture of HBS. Section 4 compares various aspects of the direct-Tcl and indirect-abstract approaches. Section 7 presents final conclusions.

2. Analysis of Hardware Build System Approaches

There exist at least more than ten build systems targeting the hardware domain. Some of them are vendor-tool-specific, while others aim to be generic, entirely abstracting away the tool used for synthesis or implementation. By looking at the architecture (analyzing how a tool internally works), existing hardware build systems can be classified into two approaches:

1. Direct-Tcl approach—The build system code is executed directly by the EDA tool during the design build flow. The build system code and the build code reside in the same domain. The build system code has access to the build code and vice versa.
2. Indirect abstract approach—The build system code produces a Tcl script, which is only later run by a proper EDA tool. The build system code and the build code reside in different domains. The build system code generates the build code, but it does not have access to it during the design build flow. The build code is not aware of the build system code and has no access to it.

2.1. Direct-Tcl Approach

In the direct-Tcl approach, the user directly writes Tcl code and uses the EDA tools' Tcl interface. The idea seems obvious. Since EDA tools already utilize Tcl as a user interface, a build system should also provide users with a Tcl interface.

Usually, the direct-Tcl approach build systems implement some wrapper commands that the user should use for primary tasks, such as, file addition, library setting, or generic/parameter setting. However, in the case of more specific tasks, the user is supposed to call the EDA tool's specific commands explicitly. It is also up to the user to maintain build

compatibility across various EDA tools when build requirements are more complex, for example, when constraints depend on synthesis results. It is often achieved with tool guard clauses. For example, an if statement block calling a specific set of commands only if the currently executed EDA tool is specific. Representatives of the direct-Tcl approach are, for example, fpga-vbs [9], vextproj [10,11], and OSVVM-Scripts [12].

Easily executing arbitrary, custom EDA tool commands is the most significant advantage of the direct-Tcl approach. As a Tcl interpreter embedded in an EDA tool executes the build system code, a user can call custom commands almost anywhere.

The direct-Tcl approach is not free of drawbacks. Some of them are inherent, and others result from the fact that direct-Tcl build systems development was stopped at some stage by their maintainers.

The first inherent drawback is that Tcl is considered an unfriendly language compared to, for example, Python. The argument is often raised by junior hardware engineers and has a very subjective nature. However, Tcl was chosen as the EDA scripting language because it was very popular when EDA tools were developed [13]. The second inherent drawback is that Tcl is not a popular language nowadays. Although there are no physical obstacles to change this trend, it is doubtful that suddenly more developers will start using Tcl instead of, for example, Python as an interpreted language. This is why this drawback is also treated as inherent.

The second group of direct-Tcl build systems drawbacks results from stopping development at a certain point. There is no direct-Tcl build system that can be used to examine the limits and bottlenecks of this idea. For example, most of the direct-Tcl build systems support only one EDA tool. OSVVM-Scripts is an exception. However, it can be used only for simulation. It does not support any synthesis or implementation tools. More example drawbacks common to the direct-Tcl build systems include the following:

1. No dependency graph generation capability.
2. Lack of or limited capability of automatic cores, targets, and testbenches detection.
3. Lack of automatic parallel testbench running.

The HBS was designed and implemented to examine the limits and bottlenecks of the direct-Tcl approach. To the author's knowledge, there is no direct-Tcl approach hardware build system supporting multiple EDA tools with features and capabilities similar to those provided by the HBS.

2.2. Indirect-Abstract Approach

The indirect-abstract approach is much more popular than the direct-Tcl approach. In this approach, the user declares various properties and configurations rather than writing Tcl code or calling EDA tools' Tcl commands directly. The main characteristic of the indirect-abstract approach is the execution of the build system code before the actual EDA tool is run. The result of the build system execution is a Tcl, Makefile, or a shell script, which is only later run by the specified EDA tool.

The most popular hardware build system utilizing the indirect-abstract approach is probably FuseSoc [14,15]. Other popular build systems include SiliconCompiler [16,17], HAMMER [18,19], Hog [20,21], hdlmake [22], orbit [23], bender [24], bazel_rules_hdl [25], flgen [26], and Blockwork [27].

The indirect-abstract hardware build systems can be subdivided into the following three subcategories based on the language and syntax they utilize:

1. Declarative—The design build flow is described in a fully declarative way utilizing markup or data serialization languages, for example, YAML. An example of the declarative indirect-abstract build system is FuseSoc.

2. Programmable—The design build flow is described using a fully functional programming language, for example, Python. An example of the programmable indirect-abstract build system is SiliconCompiler.
3. Mixed—The design build flow is partially described using the declarative way and partially programmable. An example of the mixed indirect-abstract build system is Hog.

The language and syntax used for describing build flow vary between different build systems. Some use simple data serialization formats such as YAML (for example, FuseSoc or bender) or TOML (for example, orbit). Some come with a domain-specific language (for example, flgen) or use Python (for example, SiliconCompiler, HAMMER) and Makefiles (for example, Hdlmake). Finally, some use a mix of more than one (for example, Hog). Using Python as a programming language is especially powerful, as users can use all functionalities that come with the language.

The most significant advantage of the indirect-abstract approach is the low entry threshold. The syntax of languages such as YAML, TOML, or Python is simple, which results in high legibility. Even junior engineers without extra training can understand it. Listing 1 presents a very basic core definition in the FuseSoc build system. The first line specifies the version of the description format. The “name” key specifies the vendor, library, name, and version (VLNV) identifier specified in the IP-XACT User Guide [28]. The remaining part is relatively self-explanatory.

Listing 1. Example edge detector core definition in FuseSoc build system.

CAP1=2:

```
name: vendor:library:edge_detector:1.0

filesets:
  src:
    files:
      - src/edge_detector.vhd
    file_type: vhdlSource-2008
    logical_name: hdl_library_name

targets:
  default:
    filesets:
      - src
```

The indirect-abstract approach is based on the concept that the interface to EDA tools is hidden under some common abstraction layer. Whether this is an advantage or a disadvantage is debatable. Providing a common abstract layer means that users do not have to learn the interface of multiple EDA tools. Moreover, in theory, changing the EDA tool for the compilation process should be limited to changing the EDA tool declaration solely. However, this is somewhat of a utopian view. As the practice shows, the common abstraction layer leads to the following common difficulties:

1. Missing feature implementation for some EDA tools: Let us suppose the abstraction layer declares feature F , and the build system claims support for tools $T1$ and $T2$. Although $T1$ might provide the implementation for F , $T2$ might be missing the implementation for this particular feature. In such a case, the user must add support for this feature to the build system. This task might not be trivial. The user must understand the internals of the build system, prepare a pull request, wait for a review, and potentially reiterate the implementation. Yet another approach is to modify automatically

generated build scripts before a compilation. However, this is against the whole idea of build systems. A direct-Tcl approach is free of these difficulties because the user can directly insert custom Tcl commands. There are no intermediate abstraction layers.

2. Simple tasks, such as scoping constraints to a particular module, sometimes require verbose solutions. The solution is not only more verbose but also more complex to understand. What can be solved in one or two lines of Tcl code requires more than one file. An example is discussed in Section 4.4.

All indirect-abstract hardware build systems allow “injecting” arbitrary custom Tcl commands into the automatically generated scripts. However, these commands cannot be inserted into arbitrary places. Moreover, inserting a single-line Tcl command often requires more than one line of code or even multiple files. This can be treated as proof that the indirect-abstract approach is not a golden bullet. The idea of a common abstraction layer has significant drawbacks that no one has yet found a way to fix.

Making a truly abstract hardware build system is highly unlikely in the near future. The main problem with the common abstraction layer is the fact that EDA tools were implemented before such a layer was proposed. Declaring a common interface for existing tools is a much harder task than implementing tools that conform to a formal interface.

3. Discussion of the Proposed Hardware Build System

3.1. Why the Hardware Build System Was Implemented

Characterizing and comparing direct-Tcl and indirect-abstract approaches for hardware build systems theoretically did not require HBS implementation. However, none of the existing direct-Tcl build systems were functionally close to the existing indirect-abstract build systems. In particular, none of the existing direct-Tcl build systems offered the following functionalities:

1. Support for multiple synthesis EDA tools,
2. Listing automatically detected cores and targets,
3. Automatic testbench targets detection,
4. Parallel testbench running,
5. Dependency graph generation.

The author wanted to implement a direct-Tcl build system that would functionally adhere to the existing indirect-abstract build systems. It has to be empirically verified if it is possible to have features of indirect-abstract build systems in a direct-Tcl build system.

The implemented build system (HBS) serves as a representative of the direct-Tcl approach for the comparison with indirect-abstract build systems.

3.2. Hardware Build System Requirements

The main goal of the HBS implementation was to push the direct-Tcl approach to its limits, examine its bottlenecks, and evaluate the concept of the direct-Tcl build system by comparing it with indirect-abstract approaches. However, it does not mean that HBS is solely a prototype and was designed without thoughtful requirements. This section provides a summary of HBS major requirements with extra explanation.

3.2.1. Simplicity and Ease of Use

The overhead of build system implementation and maintenance is often underrated [29,30]. McIntosh [31] reports that the build system can account for up to 31% of the code files in a project. Moreover, up to 27% of development tasks that change the source code also require build system maintenance. The study was carried out on software projects, not on hardware designs. However, due to the complexity of hardware design, it seems reasonable to assume comparable shares.

Nejati et al. in [32] report that the complexity of build system specifications is one of the top three factors complicating the review of build specifications.

The primary goal of the hardware build system proposed in this paper is simplicity. Contemporary hardware build systems utilizing the indirect-abstract approach often offer simplicity only at the initial stage. However, as the project's complexity grows, so does the complexity of using the build system.

3.2.2. Minimal User Interface

Xia et al. [33] report that even up to 21.35% of the bugs in software build systems belong to the external interface category. This means that one-fifth of build system bugs result from incorrect usage of the user interface or different behavior of the build system with different tools or platforms.

Fixing build system external interface bugs requires applying patches in the build system itself. Such a task is relatively time-consuming and might not be simple to implement. The build system user must report the problem to the build system developers or fix the bug themselves. However, this implies that the build system user is distracted from their own work. Moreover, the fix is not applied immediately. This means that the user must also find a temporary workaround. This, in turn, leads to a technical debt [34,35].

A minimal user interface has one more advantage. It allows developers to customize build systems to the project's needs. Each project is in some way unique and might require a particular approach to solve problems related to the design build. Moreover, projects usually evolve, for example, due to customer demand. Dynamic tailoring to the project requirements is a desired feature of modern build systems [36]. Similar observations were made by Neitsh et al. in [37]. Authors report that concerns from the application and implementation domains may "leak" into the build model. The build system should be adaptable enough to provide ways to respond to these concerns. Too rigid user interfaces limit such capabilities.

3.2.3. Low Number of Dependencies

Xiao et al. [38] report that dependency management is one of the top three factors contributing to the introduction of technical debt in build systems. Such a finding is not surprising. Dependencies often break in unexpected moments. In such a case, long-term quality is sacrificed for the short-term goals. Getting a project to compile successfully is more important than fixing broken build system dependencies.

One of the goals of the HBS is to minimize the probability of user distraction caused by broken dependencies. This implies that the number of dependencies should be kept as low as possible.

Moreover, a low number of dependencies potentially makes the build system more appealing for use in military designs. This is because any external dependency must be verified against potential vulnerabilities.

3.2.4. Ease of Calling Arbitrary Programs in Arbitrary Places

The ease of calling arbitrary programs in arbitrary places results from two use cases.

The first use case originates from the complexity of designing hardware systems. To reduce manual work, engineers often automatically generate parts of the design. For example, register files are automatically generated from SystemRDL [39]. In such a case, there must be a way to call an external program that generates hardware description of the register components.

Existing indirect-abstract hardware build systems allow calling external programs. However, they often require users to define a transitional so-called generator, which

increases complexity and reduces legibility. Moreover, they usually do not allow calling external programs in arbitrary places. This is further discussed in Section 6.2.

The second use case includes support for EDA tools without the Tcl interface. Such tools are often open-source simulators, for example, nvc [40] or verilator [41]. These tools have interfaces similar to the standard software compiler. The user simply calls programs with proper arguments and proper order in a shell. Being able to call arbitrary programs in arbitrary places is required to support these kinds of tools without an extra script or Makefile generation.

3.2.5. Independence from Version Control System

Some hardware build systems are tightly coupled with a particular version control system. For example, Hog is intended to be used only with git. However, build systems and version control systems are conceptually orthogonal entities. Imposing a particular version control system limits user choice. Moreover, it potentially makes incorporating a new build system for some existing projects impossible, as they might, for example, utilize Subversion instead of git.

3.3. Hardware Build System Architecture

The HBS official user manual is available in its repository [42]. The purpose of this section is to briefly describe the architecture of HBS so that the reader does not have to study the user manual.

The HBS was implemented to evaluate the limitations of the direct-Tcl approach. Nevertheless, implementing a hardware build system using this approach may be a viable strategy for the following reasons:

1. Implementing a hardware build system in Tcl allows the execution of build system code during the EDA tool flow. This, in turn, gives direct access to all EDA tool custom commands. Moreover, these custom commands can be evaluated in arbitrary places. This aspect satisfies requirements Sections 3.2.1 and 3.2.4. The complexity of existing indirect-abstract build systems for less common tasks results from executing build system code before the EDA tool flow. The build system only prepares scripts that are later run by EDA tools.
2. If the EDA tool provides the Tcl interface, then the Tcl shell is provided by the EDA tool vendor. The shell is installed during the installation of vendor tools. This implies that, in some cases, the build system user does not even have to install additional programs. This aspect satisfies requirement Section 3.2.3.
3. Executing arbitrary programs in arbitrary places in Tcl is very straightforward. There is a dedicated `exec` command for invoking subprocesses. If executing a subprocess requires prior dynamic arguments evaluation, the `exec` command call must be prepended with the `eval` command. Even in Python, invoking a subprocess is not so simple. This Tcl feature satisfies requirement Section 3.2.4.

3.3.1. Cores and Cores Detection

When the user executes `hbs`, all directories under the working directory are recursively scanned to discover all `.hbs` files (including symbolic links). Files with the `.hbs` extension are regular Tcl files that are sourced by the `hbs` script. However, before sourcing `.hbs` files, the file list is sorted so that scripts with shorter path depth are sourced first. For example, let us assume the following three `.hbs` files were found:

- `a/b/c/foo.hbs`,
- `d/bar.hbs`,
- `e/f/zaz.hbs`.

Then, they would be sourced in the following order:

1. d/bar.hbs,
2. e/f/zaz.hbs,
3. a/b/c/foo.hbs.

Such an approach allows control of when custom symbols (Tcl variables and procedures) are ready to use. For example, if the user has a custom procedure used in multiple .hbs files, then the user can create a separate utils.hbs file containing utility procedures, and place it in the project root directory. Within .hbs files, the user usually defines cores and targets, though they are free to include any valid Tcl code.

Listing 2 presents a very basic flip-flop core definition. The flip-flop core has a single target named src. The core consists of a single VHDL file.

Listing 2. Example flip-flop core definition in HBS.

```
namespace eval flip-flop {
  proc src {} {
    hbs::AddFile flip-flop.vhd
  }
  hbs::Register
}
```

To register a core, the user must explicitly call the hbs::Register procedure at the end of the core namespace. Such a mechanism helps distinguish regular Tcl namespaces from those representing core definitions.

Each core is identified by its unique path. The core path is equivalent to the namespace path in which hbs::Register is called. Using the namespace path as the core path gives the following possibilities:

1. The user can easily stick to the VLNV identifiers if required. This is presented in Listing 3. In this case, the flip-flop core path is vendor::library::flip-flop::1.0.
2. The user can define arbitrary deep core paths (limited by the Tcl shell). This is presented in Listing 4. In this case, the core path consists of seven parts.
3. The user can nest namespaces to imitate the structure of libraries and packages. This is presented in Listing 5. Three flip-flop cores are defined in the snippet. Listing 6 presents output for listing flip-flop cores.

Listing 3. Example flip-flop core definition with VLNV core path.

```
namespace eval vendor::library::flip-flop::1.0 {
  proc src {} {
    hbs::AddFile flip-flop.vhd
  }
  hbs::Register
}
```

Listing 4. Example flip-flop core definition with 7 core path parts.

```
namespace eval a::b::c::d::e::f::flip-flop {
  proc src {} {
    hbs::AddFile flip-flop.vhd
  }
  hbs::Register
}
```

Listing 5. An example .hbs file with three flip-flops core definitions

```

namespace eval lib {
  namespace eval pkg1 {
    namespace eval d-flip-flop {
      proc src {} {
        hbs::AddFile d-flip-flop.vhd
      }
      hbs::Register
    }
    namespace eval t-flip-flop {
      proc src {} {
        hbs::AddFile t-flip-flop.vhd
      }
      hbs::Register
    }
  }
  namespace eval pkg2 {
    namespace eval jk-flip-flop {
      proc src {} {
        hbs::AddFile jk-flip-flop.vhd
      }
      hbs::Register
    }
  }
}

```

Listing 6. Listing flip-flop cores output.

```

[user@host tmp]$ hbs ls-cores
lib::pkg1::d-flip-flop
lib::pkg1::t-flip-flop
lib::pkg2::jk-flip-flop

```

3.3.2. Targets and Targets Detection

The HBS automatically detects targets. Targets are all Tcl procedures defined in the scope of core namespaces (namespaces with a call to the `hbs::Register` procedure). However, to allow users to define custom utility procedures within cores, procedures with names starting with the floor character (`_`) are not treated as core targets. Listing 7 presents an example edge detector core definition. The core path is `vhdl::simple::edge-detector`. The core has three targets, `src`, `tb-sync`, and `tb-comb`, and one utility procedure, `_tb`. The `_tb` procedure was defined to share calls common for testbench targets `tb-sync` and `tb-comb`. Moreover, all target procedures are also regular Tcl procedures. Such an approach allows for calling them in arbitrary places. The `_tb` procedure calls the `src` procedure because the `edge_detector.vhd` file is required for running the testbench targets.

All targets are represented by a unique target path. The target path consists of the core path and the target name. For example, the `src` target of the edge detector has the following path:

```
vhdl::simple::edge-detector::src.
```

Listing 7. Example edge detector core definition.

```

namespace eval vhdl::simple::edge-detector {
  proc src {} {
    hbs::SetLib "simple"
  }
}

```

```

        hbs::AddFile src/edge_detector.vhd
    }
    proc _tb {top} {
        hbs::SetTool "ghdl"
        hbs::SetTop $top
        src
        hbs::SetLib ""
    }
    proc tb-sync {} {
        _tb "tb_edge_detector_sync"
        hbs::AddFile tb/tb_sync.vhd
        hbs::Run
    }
    proc tb-comb {} {
        _tb "tb_edge_detector_comb"
        hbs::AddFile tb/tb_comb.vhd
        hbs::Run
    }
    hbs::Register
}

```

3.3.3. Testbench Targets

The HBS is capable of automatically detecting testbench targets. Testbench targets are targets whose names:

1. Start with tb- or tb_ prefix,
2. End with -tb or _tb suffix,
3. Equal tb.

For example, for the edge detector from Listing 7, the hbs program detects testbench targets presented in Listing 8.

Listing 8. Testbench targets listed for example edge detector core.

```

[user@host tmp]$ hbs ls-tb
vhdl::simple::edge-detector::tb-comb
vhdl::simple::edge-detector::tb-sync

```

3.3.4. Running Targets

The HBS allows running any target of registered cores, even if the target itself has nothing to do with the hardware design. For example, running target print from Listing 9 results in the output shown in Listing 10.

Listing 9. Example print target not related to hardware build.

```

namespace eval core {
    proc print {} {
        puts "Hello!"
    }
    hbs::Register
}

```

Listing 10. Running print target result.

```

[user@host tmp]$ hbs run core::print
Hello!

```

However, in most cases, the user wants to run a target related to the flow of the set EDA tool. In such a case, instead of manually calling all required tool commands, the user can call the `hbs::Run` procedure from the core target procedure.

3.3.5. Target Parameters

As core targets are just Tcl procedures, they can have parameters. Moreover, parameters can have optional default values. Additionally, HBS allows providing command-line arguments to the run target. This is a very convenient feature in build systems. Listing 11 presents a very simplified example.

Listing 11. Example target with an optional parameter.

```
namespace eval core {
  proc target {{stage "bitstream"}} {
    puts "Running until $stage"
    # hbs::Run commented out because this is just an example.
    #hbs::Run $stage
  }
  hbs::Register
}
```

The core does not build any hardware design. However, the example shows how the build stage can be passed from the command line to an EDA tool. Listing 12 presents output from running the target with different stage parameter values.

Listing 12. Running print target result.

```
[user@host tmp]$ hbs run core::target
Running until bitstream
[user@host tmp]$ hbs run core::target synthesis
Running until synthesis
```

Another practical example of target parameters usage is setting the simulator for the testbench target from the command line or changing the top-level module. What target parameters are used for is limited only by the user's imagination and Tcl semantics.

3.3.6. Target Dependencies

In HBS, targets depend on other targets rather than on cores. Such an approach allows for fine-grained control of dependencies.

To declare target dependency, the user must call the `hbs::AddDep` procedure within the target procedure. The first argument is the dependency path. The remaining arguments are optional and are forwarded to the dependency target procedure.

To add multiple distinct dependencies, the user must call `hbs::AddDep` multiple times. The ability to pass custom arguments to a dependency was evaluated as much more advantageous than the ability to add multiple dependencies with a single `hbs::AddDep` call.

The `hbs::AddDep` internally calls the dependency procedure with the provided arguments. It also tracks dependencies, enabling the generation of a dependency graph. Within a single flow, each target procedure can be run, at most, once with a particular set of arguments. This implies that if multiple target procedures add the same dependency with the same arguments, the dependency procedure is run only once during the first `hbs::AddDep` call. To enforce a target procedure rerun, the user can always call the target procedure directly. However, enforcing target procedure rerun is usually an alert that a regular Tcl procedure shall be used instead of the core target procedure.

3.3.7. Code Generation

Code generation in the hardware design domain is ubiquitous, as it significantly speeds up implementation. For example, hardware-software co-design system-on-chip projects usually have some tool automatically generating register files [43]. Even in pure FPGA designs, it is common to generate register-transfer-level descriptions from a higher-level programming abstraction [44,45]. That is why any hardware build system needs to provide as simple mechanism for code generation as possible.

Some existing hardware build systems, for example, FuseSoc, do not allow the direct calling of an arbitrary external program in arbitrary places during the build process. Instead, the user has to define so-called generators [46]. Only then can the user call the generator within core definitions. However, such an approach has some drawbacks:

1. The generator call requires an extra layer of indirection. Generators are defined in different places than where they are used, producing decreases in the readability of the description.
2. The generator call syntax does not resemble shell command call syntax. Generators are usually regular applications that can be executed in a shell. Calling a generator within the build system using syntax similar to the shell seems natural.

In HBS, there is no formal concept of a generator. Anything can be a generator, as generators are just regular Tcl procedures. This means that generators can be target procedures (tracked by the dependency system) or core internal Tcl procedures (not tracked by the dependency system).

Listing 13 presents an example of calling an external code generator tracked by the dependency system. In actual use, the call to the shell's echo command would be replaced with a call to the proper code-generation program. Calls to the `hbs::AddFile` are commented out because no EDA tool was set.

Listing 13. Example of HBS generator tracked by dependency system.

```
namespace eval core {
    proc top {} {
        hbs::AddDep generator::gen "foo"
        puts "Adding file top.vhd"
        # hbs::AddFile top.vhd
    }
    hbs::Register
}
namespace eval generator {
    proc gen {name} {
        exec echo "Generating $name.vhd" >@ stdout
        puts "Adding file $name.vhd"
        # hbs::AddFile "$name.vhd"
    }
    hbs::Register
}
```

Listing 14 presents how to achieve the same result without tracking the generator as a dependency. This task is even more straightforward, as the user can call an external generator program directly in the target procedure.

Listing 14. Example of HBS generator not tracked by dependency system.

```
namespace eval core {
    proc top {} {
        exec echo "Generating foo.vhd" >@ stdout
```

```

puts "Adding file foo.vhd"
# hbs::AddFile "foo.vhd"

puts "Adding file top.vhd"
# hbs::AddFile top.vhd
}
hbs::Register
}

```

3.3.8. Working with High-Level Synthesis Tools

With the rapid growth of High-Level Synthesis (HLS) techniques [47,48], there is a strong requirement that any modern hardware build system should support working with IP blocks generated by HLS tools.

There are two primary ways to work with HLS tools like, for example, AMD Vitis. The first way assumes that an entire design is generated by the HLS tool. This approach is often preferred when an FPGA card is used as an acceleration card. In its paradigm, it is an equivalent of a Graphics Processing Unit (GPU) and a Compute Unified Device Architecture (CUDA) platform. In a purely compute-acceleration design, there is no need for a hardware build system, as the entire design is generated by directly calling the HLS tool's command-line interface.

The second way of working with HLS tools is to use them solely to generate an (Intellectual Property) IP block, which is later used in a hardware design compiled, for example, with AMD Vivado. The processing scope of IP cores designed with HLS spans from pure mathematical computation blocks (for example, [49]) to bus fabric control engines (for example, [50]). In the IP-only case, there is a clear distinction between the following two builds:

1. The IP block build.
2. The final hardware design build.

The IP block build is not tightly related to the final hardware design build and can be accomplished in the following two ways:

1. By a separate build script, for example, a shell script or a Makefile.
2. By calling the HLS tool directly within the .hbs file. As .hbs files are regular Tcl files, calling any system program is straightforward using the exec command.

In the case of the final hardware design build, the generated IP block must be added to the design. Because the HBS code is executed directly by the EDA tool, the user has direct access to the EDA tool custom Tcl commands. All that must be done is a call to custom Tcl commands within an .hbs file. An example is shown in Listing 15.

Listing 15. Example of HLS IP block addition for Vivado.

```

# 1. Add the path of Vitis IP folder to the project repositories
set_property ip_repo_paths path/to/the/hls_ip [current_project]
# 2. Refresh the IP catalog so Vivado indexes the new block
update_ip_catalog
# 3. Add the IP block into Block Design
create_bd_cell -type ip -vlnv xilinx.com:hls:hls_ip:1.0 hls_ip_0

```

3.3.9. Parallel Testbench Running

The HBS is capable of detecting testbench targets as described in Section 3.3.3. Moreover, HBS is capable of running testbench targets in parallel. It does not mean that HBS can speed up a single testbench by somehow making its execution parallel. This is impossible,

as it entirely depends on the simulation software used for a particular testbench. Running testbench targets in parallel means that HBS is capable of bootstrapping multiple instances of itself, with each instance responsible for running a specific testbench target.

Running testbench targets in parallel is especially useful in CI/CD pipelines as it greatly reduces the overall pipeline time.

3.3.10. Dependency Graph Generation

Whenever HBS is used to run a target, a dependency graph for that target is generated. The information on a dependency is recorded by HBS when the user calls the `hbs::AddDep` procedure within a target procedure.

The dependency graph is generated in the Graphviz DOT format (also known as the DOT language) [51], which is a plain-text representation of a graph. The advantage of such an approach is that no external dependency is required to generate the dependency graph. To visualize the graph, the user can process the generated graph text file with the `dot` program to produce a graphical representation. Figure 1 presents an example dependency graph generated for VSC8211 (Ethernet PHY) chip tester design.

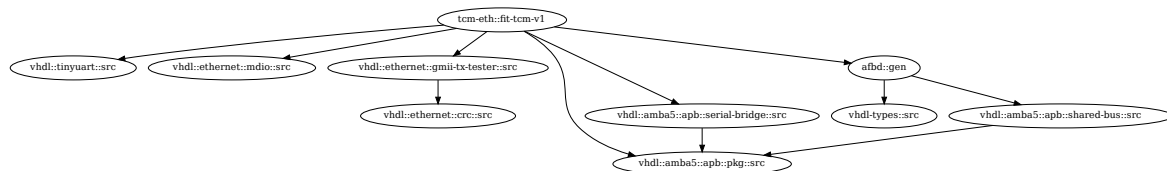


Figure 1. Example dependency graph generated by HBS.

3.3.11. Hardware Build System Commands

The implemented `hbs` program provides the user with 14 commands. Listing 16 presents the `hbs` help message. Describing particular commands is considered out of scope of the article, as it would not bring any substantive value. The commands are listed solely to present the maturity and versatility of the implemented build system. None of the existing indirect-abstract approach hardware build systems support all features present in HBS. However, some have features missing in HBS, for example, SiliconCompiler, HAMMER, or Hog.

Listing 16. HBS help message.

Usage

```
hbs <command> [arguments]
```

The command is one of:

help	Print help message
doc	Show documentation for cores
dump	Dump info about cores in Tcl dictionary format
dump-json	Dump info about cores in JSON format
graph	Output dependency graph for given target
info	Show information on hbs Tcl symbol or EDA tool
ls-cores	List cores found in .hbs files
ls-targets	List targets for given core
ls-tb	List testbench targets
run	Run given target
dry-run	Run given target without executing and evaluating commands

```

test          Run testbench targets
version       Print hbs version
where         Print where given cores are defined

```

Type 'hbs help <command>' to obtain more information about particular command.

4. Direct-Tcl and Indirect-Abstract Approaches Comparison

This section aims to characterize and compare direct-Tcl and indirect-abstract approaches for hardware build systems. The goal of the comparison is not to emerge the best hardware build system but to present differences between build systems utilizing the direct-Tcl approach (build system code executed directly by EDA tools during the design build flow) and the ones utilizing the indirect-abstract approach (build system generates Tcl script which is later run by a proper EDA tool). The best choice of the hardware build system depends on the project or user requirements.

Comparing hardware build systems is not straightforward. There are no objective numerical metrics that can be used for the build systems comparison. One such metric could be build time. However, in the case of hardware designs, the share of time consumed by EDA tools is orders of magnitude greater than the time overhead introduced by the build system code execution. This is why the build system execution time overhead is not taken into consideration at all.

Some generic software evaluation metrics could be used, such as numerical or memory complexity. However, the memory overhead introduced by build systems is negligible compared to the memory utilization of EDA tools. The numerical complexity might be important for the build system developers, but it is rather irrelevant for build system users.

This section provides a comparative analysis of direct-Tcl and indirect-abstract hardware build systems from a practical perspective. The qualitative assessment examines key operational attributes, including abstraction level, dependency overhead and its potential implications, internal code structure, constraints handling, handling EDA tools without Tcl interface, arbitrary external program calling, extra features, implementation complexity and learning difficulty.

Table 1 presents some numerical and functional features for the analyzed hardware build systems. The " $\geq N$ " signature means at least N . For build systems implemented in Python, the number of dependencies was analyzed using the pipreqs [52] tool. For build systems implemented in Rust, the number of dependencies was extracted from the Cargo.toml files. For flgen and bazel_hdl_rules, the number of dependencies was analyzed manually. Determining the exact number of dependencies (except for HBS) was not straightforward, as there might be additional runtime dependencies, hence the need for the greater-than-equal sign.

The theory of operation of both approaches was analyzed in Section 2 and is not included again in this section.

4.1. Abstraction Level

The direct-Tcl and indirect-abstract approaches present entirely different paradigms for abstraction over EDA tool interfaces.

In the direct-Tcl approach, the abstraction layer over EDA tool interfaces is limited to the common, primary actions. For example, in HBS, the following actions constitute the common abstraction layer over multiple EDA tools:

Table 1. Hardware build systems comparison (Y—yes, N—no, P—partially).

Build System	HBS	FuseSoc	SiliconCompiler	HAMMER	Hog	Hdlmake	Bender	Orbit	Blockwork	Flgen	Bazel_rules_hdl
Number of Files	1	67	552	496	99	94	73	211	87	42	360
Lines of Code	3723	6460	58,543	136,679	18,982	8494	12,026	46,690	8610	3621	42,743
External Dependencies	2	>12	>37	>9	>3	>5	>23	>20	>16	>10	>7
Implementation Language	Tcl	Python	Python	Python	Shell/Tcl	Python	Rust	Rust	Python	Ruby	Starlark
Introduced File Extensions	1	1	0	0	4	1	2	1	0	1	0
Introduced File Syntaxes	0	0	0	0	1	0	0	0	0	1	1
Dependency Graph Generation	Y	N	Y	N	N	N	Y	Y	N	N	N
Build System Code Executed During Design Build	Y	N	N	N	P	N	N	N	N	N	N
Build System Code Executed Directly by EDA Tool	Y	N	N	N	P	N	N	N	N	N	N
Tied to Single Revision System	N	N	N	N	Y	N	N	N	N	N	N
Requires Specific Directory Structure	N	N	N	N	Y	N	N	N	N	N	N
Capable of Building Designs	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Y
Requires Containerization	N	N	N	N	N	N	N	N	Y	N	N
Automatic Testbench Running	Y	N	N	N	N	N	N	N	N	N	N
Distributed execution of compilation flows	N	N	Y	N	N	N	N	N	N	N	N

1. Target device setting. All synthesis EDA tools require information on the target device. This is why setting the device became part of the common abstraction layer.
2. File addition—this includes support for adding files of all formats supported by a given EDA tool.
3. Library setting—setting HDL file library.
4. HDL standard setting—setting HDL file standard revision. The build system has to manage this because some tools cannot analyze different design units with different standard revisions, for example, nvc [53]. In such a case, the build system must decide which common standard revision to use for analyzing all HDL files.
5. Dependency specification—this is the core feature of any build system.
6. Generics/parameter setting—configuring parametric designs must be an inherent feature of any hardware build system.
7. Design top module setting—all EDA tools performing simulation or synthesis require information about the top module.
8. Exit severity setting—testbenches are designed to exit with an error code when a specific severity message occurs. This is independent of the simulator being used; thus, it should be hidden under the build system abstraction.

In the direct-Tcl approach, the build system intentionally exposes EDA tool-specific quirks so that the user has to resolve them explicitly. This is usually achieved with tool guard clauses. Listing 17 presents an example of such a guard. The snippet presents a helper testbench procedure definition for the AMBA 5 APB crossbar. If the tool is set to the nvc simulator, an extra argument “--dump-arrays” must be passed to the simulator to dump array signals to the waveform file. This can be seen as a peculiarity of the nvc simulator.

Listing 17. An example of tool guard clause in HBS (direct-Tcl approach).

```
proc _tb {top} {
    hbs::AddDep vhd1::amba5::apb::checker::src
    hbs::AddDep vhd1::amba5::apb::bfm::src
    hbs::AddDep vhd1::amba5::apb::mock-completer::src
    if {$hbs::Tool eq "nvc"} {
        hbs::AddPreSimCb hbs::SetArgSuffix "--dump-arrays"
    }
    hbs::SetTop $top
    hbs::AddDep $hbs::ThisCorePath\::src
}
```

In the indirect-abstract approach, the main purpose of the abstraction layer is to provide a great, unified abstraction over EDA tools. As such, the abstraction layer covers many more aspects than the abstraction layer in the direct-Tcl approach. However, maximizing the abstraction layer is not the goal of the indirect-abstract approach; the goal is to unify the user experience.

Instead of exposing EDA tools' peculiarities to the user, the indirect-abstract approach tries to hide them under the abstraction layer. For example, the nvc peculiarity of not dumping arrays signals by default in the indirect-abstract approach is solved in such a way that build systems automatically add the “--dump-arrays” argument when the simulation is run.

4.2. Dependency Overhead and Its Potential Implications

The term “dependency” in the following context means a program, library, or operating system package that must be explicitly installed by the user to utilize all features of the analyzed build system. Any tools installed automatically alongside EDA tools are not treated as dependencies.

Hardware build systems representing the direct-Tcl approach tend to have fewer dependencies. For example, the implemented HBS has only the following two dependencies:

1. Tcl shell (mandatory)—required to run the core logic of the build system.
2. Graphviz (optional)—required for generating a core dependency graph in a graphical format.

None of the existing hardware build systems representing the indirect-abstract approach, capable of similar features, have so few dependencies.

The explanation for the lower number of dependencies in build systems that use the direct-Tcl approach is straightforward. They rely on libraries and programs that are installed during EDA tool installation. On the other hand, build systems utilizing the indirect-abstract approach often rely on dependencies installed manually via package managers.

Nevertheless, a pure number of dependencies might not be a good metric for comparison. A project could have a single dependency that suffers from multiple issues, and another project could have hundreds of well-maintained dependencies.

What might be more important from the user's point of view is stability. A build system is expected to work across many systems with a wide range of long-term support.

The ease of initial setup of the build system might also be relevant to the user. The direct-Tcl approach with fewer dependencies is probably less susceptible to stability issues. However, overall stability depends on multiple factors, such as server stability, OS stability, network reliability, CI/CD complexity, etc.

4.3. Internal Code Structure

The internal code structure was analyzed in terms of the number of files, lines of code, introduced file syntaxes and extensions, and supported features. The following types of files were excluded from analysis: author files, news files, contributors files, license files, readme files, tests files, and example files. The goal of the analysis is to provide raw facts.

In Table 1, the introduced file extensions are the number of custom file extensions used by a build system. The introduced file syntaxes are the number of custom syntaxes used by a build system. For example, FuseSoc scans for files with the `.core` extension. However, the `.core` files have a standard YAML syntax. This is why the number of introduced file extensions equals 1 but the number of introduced file syntaxes equals 0.

An extra context has to be introduced to the table, as different tools support different features and EDA tools. Direct-Tcl build systems (except HBS) are not included in the table. This is because they support only one EDA tool, and they do not support automatic testbench detection and dependency graph generation. OSVVM-Script is not included because it targets only simulation tools.

The FuseSoc build tool internally uses the external Edalize [54] library to interact with EDA tools. Including Edalize, the number of files for FuseSoc equals 215, and the number of lines of code equals 19,788.

The total number of lines of code of any hardware build system depends heavily on the number of supported EDA tools. FuseSoc supports the greatest number of tools, almost 40. HBS currently supports the following seven tools: AMD Vivado, AMD xsim, GHDL [55], GOWIN IDE, Intel/Altera Quartus, nvc, and Questa. The most complex one is Vivado, in which full support requires roughly 190 lines of code. However, supporting the nvc simulator required roughly 170 lines of code. Assuming that HBS will one day support 40 EDA tools, the number of lines of code should remain below 10,000.

Tools following the direct-Tcl approach tend to have a more monolithic code structure, fewer files, and fewer lines of code. The code for handling EDA tools is often placed in a single file. The lower number of lines of code results from the direct access to the EDA tool

commands. There is no need for wrapper code. They prefer shared global variables, which makes build logic easier to follow but also more bug-prone.

Tools following the indirect-abstract approach tend to have a more modular code structure. They have more files and a greater number of lines of code. The code for handling specific EDA tools is organized into modules in separate files. The encapsulation makes the code more bug-resistant, but the build logic is slightly harder to follow.

4.4. Constraints Handling

Building any hardware design (FPGA or ASIC) without constraints is impossible. Constrains usually define clocks, path delays, timing relations, and physical properties, for example, pin locations, drive strength or voltage level. Any hardware build system must support various constraint types and scenarios. The goal of this subsection is to present differences in constraint handling from the user perspective of the build system.

In the case of a simple constraint scenario with global constraints, there is no difference between the direct-Tcl and indirect-abstract approaches. The user simply defines constraints in a supported file format (e.g., .sdc, .xdc, or .tcl) and adds the file to the project or build flow. Based on the file extension, build systems decide how to handle the file. However, there is a noticeable difference when a project requires constraints scoped to modules and cells, or when constraints are added optionally.

5. Scoping Constraints to Module

Let us analyze a simple example. The engineer implements a module and wants to scope some constraints to this module. As an example, a two-stage flip-flop clock domain crossing synchronizer is used. The source code of the synchronizer is not relevant to the analysis. The false path is set as a constraint, although the max delay is also frequently used.

Listing 18 presents the synchronizer core definition in FuseSoc, which represents the declarative indirect-abstract subcategory.

Listing 18. Cross-domain crossing synchronizer core definition in FuseSoc.

CAP1=2:

```
name: vendor:library:synchronizer

filesets:
  src:
    files:
      - synchronizer.vhd
    file_type: vhdlSource-2008
  vivado_constr:
    files:
      - vivado-constr.xdc: {file_type: xdc}
  tcl:
    files:
      - constr.tcl: {file_type: tclSource}

targets:
  default:
    filesets:
      - src
      - "tool_vivado? (vivado_constr)"
      - tcl
```

Listing 19 presents the content of the `vivado-constr.xdc` file, and Listing 20 presents the content of the `constr.tcl` file.

Listing 19. Content of the `vivado-constr.xdc` file.

```
set_false_path -to [get_pins s_0_reg[*]/D]
```

Listing 20. Content of the `constr.tcl` file for FuseSoc.

```
if {[string match "Vivado*" [version]]} {
    set_property SCOPED_TO_REF Synchronizer \
        [get_files vivado-constr.xdc]
} else {
    error \
        "Synchronizer entity misses constraint file for your EDA tool"
}
```

In FuseSoc, to accomplish the task, the user must provide four files:

1. `synchronizer.vhd`—synchronizer HDL description,
2. `vivado-constr.xdc`—false path Vivado constraint for synchronizer,
3. `constr.tcl`—Tcl script scoping constraint to synchronizer module,
4. `synchronizer.core`—synchronizer core definition.

The number of required files (4) is greater than the number of logical elements (3). The user describes the module in HDL, defines the core for the build system, and adds extra constraints. The number of logical elements equals three, and the number of files equals four. The logic of the core definition is more complex to follow, as the user has to switch between files for a very short snippet of code. This may lead to a decrease in productivity [56].

Listing 21 presents the same core definition but in the direct-Tcl approach. The number of required files is three and equals the number of logical elements. The `constr.tcl` file could be eliminated because the code of the build system is executed directly by the EDA tool. Even the error message is improved. It is possible to report which EDA tool the constraint file definition is missing for. Again, this is possible because the build system code is executed directly by the EDA tool.

In the case of the indirect-abstract approach, build scripts are prepared before the EDA tool is run. Within the `constr.tcl` script, there is no access to the build system context or variables, as the build system code execution is already finished.

Listing 21. Cross domain crossing synchronizer core definition in HBS (direct-Tcl approach).

```
namespace eval vendor::library::synchronizer {
    proc src {} {
        hbs::AddFile synchronizer.vhd

        if {$hbs::Tool == "vivado-prj"} {
            hbs::AddFile vivado-constr.xdc
            set_property SCOPED_TO_REF Synchronizer \
                [get_files vivado-constr.xdc]
        } else {
            error \
                "Synchronizer entity misses constraint file for $hbs::Tool"
        }
    }
}
```

```

    hbs::Register
}

```

In the case of the programmable indirect-abstract subcategory, the solution depends on the tool. For example, in hdlmake, the intermediate Tcl script is also required to scope a constraint file to a module. The same script is also required for the SiliconCompiler. Hog, for example, instead of a Tcl script, requires a custom constraint list file (a file with the .con extension) with paths to constraint files and their property declarations. Moreover, the only way to opt out of the default constraints is to comment out the constraint lines in the constraint list file. To the author's knowledge, none of the existing indirect-abstract build systems utilize the concept of direct injection of Tcl code into the build script. They all need one additional file (a Tcl script or a tool-specific file) for scoping constraints.

6. Scoping Constraints to Cells and Optional Constraints

Let us analyze a more complex example. Namely, the Advanced Peripheral Bus Clock Domain Crossing bridge (APB CDC bridge). Such a bridge requires some timing constraints due to the clock domain crossing. There are a few possible ways to define valid constraints for the APB CDC bridge. The core designer may want to provide some default constraints. However, the user should also be able to provide their own constraints.

One possible constraint strategy is to set a false path for handshake signals and a maximum delay for bus signal crossing domains. The max delay value should equal the period of the destination clock. The false path constraints can be safely scoped to the APB CDC bridge module. However, because there may be multiple APB CDC bridges in the design crossing between various clock domains, the max delay constraints should be scoped to cells. An alternative solution is to pick the shortest clock period as the value for the max delay constraint for all bridges. However, such an approach leads to design overconstraint and imposes unnecessary requirements on the router.

Listing 22 shows how APB CDC bridge constraints are handled in the HBS, which represents the direct-Tcl approach. The code is concise and located within a single procedure. The user can pass the -no-constr flag to the target to omit adding the default constraints. The flag is checked with a simple if statement. Moreover, if an invalid argument is passed, an error with a precise message is returned. The apb-cdc-bridge-ref.xdc and apb-cdc-bridge-cell.tcl constraint files are not presented as their content is irrelevant. The first one has constraints scoped to the module; the second one has code that applies constraints scoped to cells.

Listing 22. Advanced Peripheral Bus Clock Domain Crossing bridge target definition in HBS (direct-Tcl).

```

# Use -no-constr flag to omit adding the default constraints.
proc src {args} {
    hbs::AddDep vhd1::amba5::apb::pkg::src
    hbs::SetLib "lapb"
    hbs::AddFile cdc-bridge.vhd

    set argCount [llength $args]
    if {$argCount == 1} {
        set arg [lindex $args 0]
        if {$arg eq "-no-constr"} {
            return
        } else {
            error "$hbs::ThisTargetPath: invalid arg '$arg', " \
                "valid args are: '-no-constr'"
        }
    }
}

```

```

} elseif {$argCount > 1} {
    error \
        "$hbs::ThisTargetPath: invalid arg count $argCount, " \
        "expected at most 1"
}

if {$hbs::Tool eq "vivado-prj"} {
    hbs::AddFile vivado/apb-cdc-bridge-ref.xdc
    set_property SCOPED_TO_REF APB_CDC_Bridge \
        [get_files apb-cdc-bridge-ref.xdc]
    hbs::AddFile vivado/apb-cdc-bridge-cell.tcl
    set_property USED_IN_SYNTHESIS false \
        [get_files apb-cdc-bridge-cell.tcl]
} else {
    error "$hbs::ThisCore misses constraints for $hbs::Tool"
}
}

```

How applying the same constraints in the indirect-abstract approach looks depends on the tool subcategory. For example, in FuseSoc, which represents the declarative subcategory, applying the module and cell-scoped constraints would be similar to what was presented in Listings 18 and 20 for the CDC synchronizer. Again, one more Tcl file would be required. The optionality of the constraints would be handled using the flags mechanism [57]. While the flag mechanism allows conditional inclusion of files, it does not allow arbitrary actions to be executed due to its declarative nature.

In hdlmake, the same core description would require two Manifest.py files. The first one with constraints included, and the second one without. The user would pick a proper Manifest.py file when declaring dependency on the core. With just a single boolean parameter, this approach does not seem complex. However, with multiple parameters, the number of Manifest.py files would equal the number of possible combinations.

In the programmable subcategory of the indirect-abstract approach, the complexity of constraint handling is lower. In this subcategory, the solution is very similar to the one present in the direct-Tcl approach. However, instead of directly injecting the Tcl code into the generated Tcl build script, the user must place it in a separate Tcl file. There is only minor overhead resulting from the following:

1. Build system programming language Application Programming Interface (API) call,
2. Build logic fragmentation due to the requirement of putting the Tcl code in a separate file.

6.1. Handling EDA Tools Without Tcl Interface

Not all EDA tools have a Tcl interface. For example, free and open-source simulators, such as Verilator [58] or GHDL, have a shell interface. The user simply calls programs within the shell, providing appropriate arguments.

Handling EDA tools without the Tcl interface in the direct-Tcl approach is straightforward. The Tcl language makes it easy to call any external program using the `exec` command. Listing 23 presents how the elaboration stage for the GHDL simulator is implemented in HBS. The command to be executed is constructed directly as a string to be passed to the shell. The code related to the working directory change and error checking takes more lines than the code creating the actual command. The `hbs::Exec` procedure is a simple wrapper around the Tcl `exec` command, which allows printing of the command to the standard output in the dry-run mode.

Listing 23. GHDL elaboration stage implementation in HBS (direct-Tcl approach).

```

proc elaborate {} {
    set workDir [pwd]
    cd $hbs::RunTargetBuildDir

    set cmd "ghdl -e $hbs::ArgPrefix --std=[hbs::ghdl::std] " \
           "--workdir=work $hbs::ghdl::libs $hbs::ArgSuffix $hbs::Top"
    set err [hbs::Exec $cmd]
    if {$err} {
        hbs::panic "$hbs::Top elaboration failed with exit status $err"
    }

    cd $workDir
}

```

In the indirect-abstract approach, handling EDA tools without the Tcl interface is implemented differently. In the indirect-abstract approach, the build system code is run before the build code. This implies that the build system code must generate the build code that is only later run. The build code is usually generated in the form of a Makefile or shell script. Compared to the direct-Tcl approach, there are two essential differences:

1. The generated build code script can be easily reused without rerunning the build system code.
2. The build system code logic might be slightly harder to follow. This is because the data for the script templates is usually not prepared in a linear way that corresponds to the build flow. Conversely, in the direct-Tcl approach, the logic must be linear because the build code is executed during the build system code execution.

6.2. Arbitrary External Program Calling

Arbitrary external program calling is a very useful feature of modern hardware build systems. The word “arbitrary” means two things:

1. Arbitrary program—the user should be able to call any program available on the operating system.
2. Arbitrary place—the user should be able to call programs in any place within the design tool flow.

A practical example of an arbitrary program is a code generator. Code generation in the hardware design domain is ubiquitous, as it significantly speeds up implementation.

A practical example of an arbitrary place is report analysis. A user may want to analyze the synthesis stage results before starting the implementation stage. However, the user may want to implement the reports parsing logic in a Perl or Python script. In such a case, there must be a way to call the custom script within the Tcl build script after the synthesis.

Meeting the two arbitrariness requirements (program and place) in the direct-Tcl approach is straightforward. This is because of the following reasons:

1. Tcl has built-in exec command for executing external programs.
2. The build system code is executed directly by the EDA tool.
3. The build system code is executed during the design build flow.

Listing 24 presents an example of how to execute a command after the synthesis step in HBS.

Listing 24. Post synthesis callback example in HBS (direct-Tcl).

```

proc myPostSynthCb {} {
    set err [catch {eval exec "./analyze-synth-reports.py"} output]
    if {$output ne ""} {
        puts "synthesis results are not satisfactory: $output"
        exit 1
    }
}

hbs::AddPostSynthCb myPostSynthCb

```

In HBS, each stage has optional pre- and post-callbacks. Figure 2 shows the structure of the tool flow for the Vivado project mode.

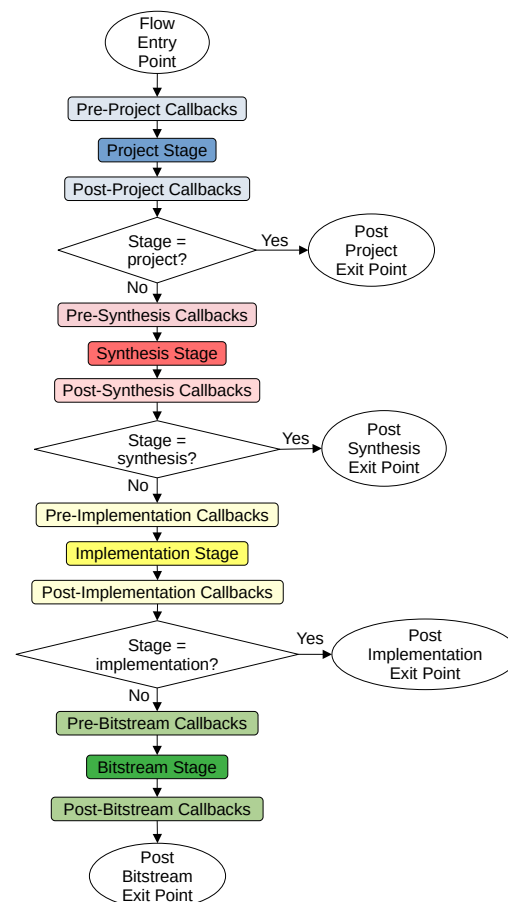


Figure 2. Vivado project mode tool flow.

There are four stages: project, synthesis, implementation, and bitstream. The number of callbacks is unlimited. Callbacks in any pre- or post-stage are executed in the order users add them. The user adds a callback by calling a dedicated HBS API procedure. For example, to add a post-synthesis callback, the user can call the `hbs::AddPostSynthCb` procedure. Callbacks can be also added with custom argument values.

While most existing indirect-abstract hardware build systems allow the user to call any third-party programs, the tools representing the declarative subcategory do not allow calling them in arbitrary places. For example, in FuseSoc, the user has to define so-called generators [46]. Only then can the user call the generator within core definitions. However, such an approach has some drawbacks:

1. The generator call requires an extra layer of indirection. Generators are defined in different places from where they are used, which reduces the readability of the description.
2. The generator call syntax does not resemble the shell command call syntax. Generators are usually regular applications that can be executed in a shell. Calling a generator within the build system using syntax similar to the shell seems natural.
3. Generators can be executed only before the EDA tool flow.

In the case of build systems representing the programmable indirect-abstract subcategory, meeting the two arbitrariness requirements is also straightforward. However:

1. it requires some Python wrapper code,
2. the code must be placed in a separate script file.

Listing 25 shows how to add a post-synthesis script in SiliconCompiler. There are dedicated Python API functions, so the procedure is very concise. Compared to the direct-Tcl approach, there is one more indirection layer, as the actual Tcl code must be placed in a separate file, which leads to the build logic fragmentation.

Listing 25. Post-synthesis callback example in SiliconCompiler (indirect-abstract).

```
SynthesisTask.find_task(project).add_postscript(
    "path/to/post-synth-script.tcl"
)
```

Listing 26 presents how to execute a command after the synthesis stage in hdlmake. The snippet is a part of a Manifest.py file. hdlmake, like SiliconCompiler, requires the code to be placed in a separate Tcl file. Moreover, hdlmake closes the project after every step. If the evaluated code requires access to the build context, for example, synthesis results, the user must explicitly open the project in the script.

Listing 26. Post-synthesis program execution example in hdlmake (indirect-abstract).

```
syn_post_synthesize_cmd = "$(TCL_INTERPRETER) user_tcl_script.tcl;"
```

6.3. Extra Features Support

Modern build systems are expected to do much more than produce the final production or programming files. In terms of hardware build systems, for example, the following extra features are desired:

1. Automatic testbench targets detection,
2. Parallel testbench running,
3. Core dependency graph generation,
4. Distributed development and distributed execution of compilation flows.

None of the existing direct-Tcl hardware build systems support feature number 4. Moreover, HBS is the only direct-Tcl build system supporting features 1, 2, and 3.

In the indirect-abstract class, only SiliconCompiler, bender, and orbit support feature number 3. The SiliconCompiler is the only existing hardware build system with built-in support for distributed development and distributed execution of compilation flows.

SiliconCompiler and HAMMER have the most extensive support for ASIC and VLSI designs. Both were designed from the ground up for that purpose and used for commercial projects in this domain. However, both of them have the greatest internal complexity.

It is also important to note the following:

- The bender is only a dependency management tool. It provides a way to define dependencies among cores, execute unit tests, and verify that the source files are valid

inputs for various simulation and synthesis tools. However, it does not run the EDA tool for building designs.

- The flgen is just a file list generator. It does not run EDA tools for building designs.

6.4. Extra Features Implementation Complexity

While hardware build system users are mostly interested in extra-feature support (Section 6.3), hardware build system developers might also be interested in the implementation complexity of extra features.

Implementing extra, non-build-related features in the direct-Tcl approach is more complex than in the indirect-abstract approach. This state results directly from the use of the Tcl language. While multi-threading in Tcl is possible, the Tcl was not designed with multi-threading as one of its primary goals. This explicitly impacts support for parallel testbench execution or distributed compilation. Moreover, Tcl has fewer third-party libraries that could help implement additional features.

Yet another important factor is the fact that direct-Tcl build systems are executed by the Tcl interpreters provided by the EDA tools. Each version of the EDA tool is shipped with a fixed version of the Tcl interpreter. Modern third-party Tcl libraries might require a higher version of the Tcl interpreter, which might potentially block them from being used within EDA tools. In the indirect-abstract approach, the build system infrastructure (tools and libraries) is managed independently from the EDA tools.

The direct-Tcl approach tools tend to have more code related to the implementation of extra features because any feature requiring parallelism has to be implemented asynchronously using the `filevent` and `vwait` mechanisms. Moreover, the code has a non-linear structure and is spread across multiple locations. An alternative approach to mitigate the mentioned deficiencies in the direct-Tcl approach is to implement the extra features in a wrapper program or script written in another programming language. However, this leads to splitting the build system's code logic across multiple languages, which might increase cognitive complexity. Hardware build systems that follow the indirect-abstract approach are usually free of the mentioned drawbacks, as they only generate Tcl code and are entirely written in languages more suitable for implementing such extra features.

6.5. Learning Difficulty

The direct-Tcl approach might be harder to learn for less experienced engineers. There are two reasons for such a state:

1. The Tcl language is not considered to be the most user-friendly.
2. The user must be familiar with the EDA tool commands in the case of less common tasks, for example, working with block designs or generating vendor cores using the Tcl interface instead of the IP wizard.

The indirect-abstract approach has a lower entry point. In the case of the declarative subcategory, the user must only know basic formats, such as YAML or TOML, and a few build system keywords. In the case of the programmable subcategory, the language of choice is Python, which is commonly considered easier to learn than Tcl.

It is worth noting that, in contemporary contexts, the complexity associated with programming languages and EDA tools is effectively mitigated through the application of artificial intelligence (AI) techniques [59,60].

7. Conclusions

Table 2 summarizes characteristics of direct-Tcl and indirect-abstract approaches for hardware build systems.

Table 2. Direct-Tcl and indirect abstract hardware build system approaches comparison summary.

Approach	Direct-Tcl	Indirect-Abstract
Theory of operation	The build system code is executed during the design build flow. The build system code is executed directly by the EDA tool. The build system code and build code reside in the same domain.	The build system code is executed before the design build flow. A build system generates Tcl or shell script that is only later run by the EDA tool. The build system code and build code reside in different domains.
Dependency overhead and its implications	Fewer dependencies. Initial environment setup requires fewer installations. Lower probability of stability problems on long-term support distributions.	More dependencies. Initial environment setup requires more actions. Slightly higher probability of stability problems on long-term support distributions.
Internal code structure	Monolithic. The code containing logic for EDA tools handling is placed within a single file. Prefers shared global variables which makes build logic easier to follow but also bug-prone.	Modular. The code for handling particular EDA tools is placed within modules in separate files. The encapsulation makes the code more bug-resistant, but the build logic is slightly harder to follow.
Abstraction level	Low. The common abstraction layer is limited to primary tasks. For example, file faddition, library setting, generic/parameter setting. Intentionally exposes EDA tool peculiarities to the user.	High. The common abstraction layer tries to cover all areas and hide the peculiarities of different EDA tools. Makes porting a design to a different EDA tool easier.
Constraints handling	Simple. As the build code is executed directly by the EDA tool during the build system code execution, there is direct access to all constraint commands. Optional constraints addition can be handled with a simple if statement.	Depends on the subcategory. In the case of the declarative subcategory, some less common constraint scenarios are more complex to achieve and require indirection layers. In the case of the programmable subcategory, constraint handling requires only a small amount of wrapper code.
Handling EDA tools without Tcl interface	Straightforward. These kinds of tools are directly called using the exec command or catch-eval-exec sequence if error handling is required. Build rerun requires build system code rerun.	Slightly more complex. EDA tools without the Tcl interface are handled by generating Makefile or shell scripts. The generated build code scripts can be easily reused without rerunning the build system code.
Arbitrary external program calling	Straightforward. User simply calls Tcl exec command if error catching is not required or catch-eval-exec sequence if error catching is required. Programs can be easily called between any EDA tool flow stages.	Depends on the core description file format. In the case of markup or data serialization languages, calling external programs is complex. In the case of, for example, Python, calling external programs is simple, although it still requires slightly more code than in Tcl.
Extra features support	Only HBS supports automatic testbench detection, parallel testbench running, and dependency graph generation. No tool natively supports distributed execution of compilation flows.	Highly depends on the tool. For example, SiliconCompiler is the only tool natively supporting the distributed execution of compilation flows. Only SiliconCompiler, bender and orbit support core dependency graph generation.
Extra feature implementation complexity	Higher. Extra features, for example, parallel testbench running, require more code. Moreover, the code is asynchronous and has a non-linear structure. The build system code gets fragmented.	Lower. Build systems representing this approach do not have to be implemented in Tcl. Most of them use Python, which makes implementing extra features easier and does not split the build system code logic.
Learning difficulty	Higher. The user must directly write Tcl. Moreover, the user has to be familiar with EDA tools commands in the case of less common tasks, for example, working with block designs.	Lower. Markup, data serialization, and Python syntaxes are much simpler and easier to read even for newcomers.

The taxonomical data presented in the table can serve as a strategic framework for selecting an optimal hardware build system based on specific project requirements and organizational constraints. For example, in the cases of massive-scale ASIC projects that scale to thousands of developers and even millions of servers, the indirect-abstract approach is currently the only viable solution. Valid choices might be SiliconCompiler or HAMMER, as both provide the necessary abstraction layers to manage this complexity. For small-to-medium projects with a low entry threshold, a pure declarative, indirect-abstract hardware build system might be a good choice, such as FuseSoc. In the case of tight coupling with Git and CI/CD, Hog might be the best choice. For medium-to-large projects targeting a single-vendor ecosystem, a direct-Tcl approach (e.g., HBS) is often superior. The direct-Tcl approach is particularly effective for teams with deep expertise in EDA tool command interfaces, where the steeper learning curve is offset by granular control over the implementation flow.

Beyond selection, the comparative analysis in Table 2 can provide a roadmap for developing next-generation hardware build systems. By identifying the inherent bottlenecks and architectural weaknesses of existing approaches, developers can proactively implement mitigation strategies. Understanding these trade-offs a priori allows for the design of more resilient, scalable hardware build systems that address the limitations of current direct-Tcl and indirect-abstract tools.

Funding: This research received no external funding. The APC was funded by the Warsaw University of Technology.

Data Availability Statement: The original data presented in the study are openly available in HBS repository at <https://github.com/m-kru/hbs> (accessed on 1 July 2026).

Conflicts of Interest: The author declares no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
AMD	Advanced Micro Devices
ASIC	Application-Specific Integrated Circuit
CDC	Clock Domain Crossing
CI/DC	Continuous Integration/Continuous Deployment
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
GNU	GNU's not Unix
HBS	Hardware Build System
HDL	Hardware Description Language
HLS	High-Level Synthesis
IDE	Integrated Development Environment
IP	Intellectual Property
OS	Operating System
VLSI	Very-Large-Scale Integration
VLNV	Vendor Library Name Version

References

1. Smith, P. *Software Build Systems: Principles and Experience*; Addison-Wesley Professional: Boston, MA, USA 2011.
2. Mokhov, A.; Mitchell, N.; Peyton Jones, S. Build systems à la carte: Theory and practice. *J. Funct. Program.* **2020**, *30*, e11. <https://doi.org/10.1017/S0956796820000088>.

3. Free Software Foundation. GNU Make. Available online: <https://www.gnu.org/software/make> (accessed on 10 March 2026).
4. CMake. CMake: A Powerful Software Build System. 2025. Available online: <https://cmake.org/> (accessed on 10 March 2026).
5. Evan Martin. Ninja. 2025. Available online: <https://ninja-build.org/> (accessed on 10 March 2026).
6. Mithro, E. Software Inspired IC Hardware Workflows Using Bazel. In Proceedings of the 4th Workshop on Open-Source EDA Technology (WOSET), Virtual, 18 November 2021.
7. Google. Bazel. 2025. Available online: <https://bazel.build/> (accessed on 10 March 2026).
8. Feldman, S.I. Make—A Program for Maintaining Computer Programs. *Softw. Pract. Exper.* **1979**, *9*, 255–265. <https://doi.org/10.1002/spe.4380090402>.
9. Braun, A. MLE FPGA Buildsystem for AMD Vivado TM. 2025. Available online: <https://github.com/missinglinkelectronics/fpga-vbs> (accessed on 9 March 2026).
10. Zabołotny, W.M. Vextproj. 2025. Available online: <https://github.com/wzab/vextproj> (accessed on 9 March 2026).
11. Zabołotny, W.M. Version control friendly project management system for FPGA designs. In Proceedings of the 38th Wilga Symposium on Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2016, Wilga, Poland, 29 May–6 June 2016; Volume 10031. <https://doi.org/10.1117/12.2247944>.
12. Lewis, J. OSVVM-Scripts. 2025. Available online: <https://github.com/OSVVM/OSVVM-Scripts> (accessed on 10 March 2026).
13. Chen, P.; Kirkpatrick, D.; Keutzer, K. Scripting for EDA tools: A case study. In Proceedings of the IEEE 2nd International Symposium on Quality Electronic Design, San Jose, CA, USA, 26–28 March 2001; pp. 87–93. <https://doi.org/10.1109/ISQED.2001.915211>.
14. Kindgren, O. Invited Paper: A Scalable Approach to IP Management with FuseSoC. 2019. Available online: https://fusesoc.net/osda19_fusesoc.pdf (accessed on 10 March 2026).
15. Kindgren, O. FuseSoC. 2025. Available online: <https://github.com/olofk/fusesoc> (accessed on 10 March 2026).
16. Olofsson, A.; Ransohoff, W.; Moroze, N. A distributed approach to silicon compilation: Invited. In Proceedings of the 59th ACM/IEEE Design Automation Conference, New York, NY, USA, 10–14 July 2022; DAC '22; pp. 1343–1346. <https://doi.org/10.1145/3489517.3530673>.
17. SiliconCompiler Project. SiliconCompiler. 2025. Available online: <https://github.com/siliconcompiler/siliconcompiler> (accessed on 11 March 2026).
18. Liew, H.; Grubb, D.; Wright, J.; Schmidt, C.; Krzysztofowicz, N.; Izraelevitz, A.; Wang, E.; Asanović, K.; Bachrach, J.; Nikolić, B. Hammer: A modular and reusable physical design flow tool: Invited. In Proceedings of the 59th ACM/IEEE Design Automation Conference, New York, NY, USA, 10–14 July 2022; DAC '22; pp. 1335–1338. <https://doi.org/10.1145/3489517.3530672>.
19. Berkeley Architecture Research. HAMMER. 2025. Available online: <https://github.com/ucb-bar/hammer> (accessed on 11 March 2026).
20. Biesuz, N.; Camplani, A.; Cieri, D.; Giangiacomi, N.; Gonnella, F.; Peck, A. Hog (HDL on git): A collaborative management tool to handle git-based HDL repository. *J. Instrum.* **2021**, *16*, T04006. <https://doi.org/10.1088/1748-0221/16/04/T04006>.
21. Gonnella, F.; Cieri, D.; Giangiacomi, N.; Biesuz, N.; Camplani, A.; Peck, A.; Loustau de Linares, G. Hog (HDL-on-Git). 2025. Available online: <https://gitlab.com/hog-cern/Hog> (accessed on 11 March 2026).
22. CERN Open Hardware Repository. Hdlmake. 2025. Available online: <https://ohwr.org/projects/hdl-make/> (accessed on 12 March 2026).
23. Ruskin, C. Orbit. 2025. Available online: <https://github.com/chaseruskin/orbit> (accessed on 11 March 2026).
24. PULP Platform. Bender. 2025. Available online: <https://github.com/pulp-platform/bender> (accessed on 12 March 2026).
25. The Hardware Description Languages GitHub Project. Bazel_rules_hdl. 2025. Available online: https://github.com/hdl/bazel_rules_hdl (accessed on 12 March 2026).
26. Ishitani, T. Flgen. 2025. Available online: <https://github.com/pezy-computing/flgen> (accessed on 12 March 2026).
27. Blockwork EDA. Blockwork. 2025. Available online: <https://github.com/blockwork-eda/blockwork> (accessed on 12 March 2026).
28. Accellera IP-XACT Working Group. IP-XACT User Guide. 2018. Available online: https://www.accellera.org/images/downloads/standards/ip-xact/IP-XACT_User_Guide_2018-02-16.pdf (accessed on 14 March 2026).
29. Nejati, M.; Alfadel, M.; McIntosh, S. Understanding the Implications of Changes to Build Systems. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 27 October–1 November 2024; ASE '24; pp. 1421–1433. <https://doi.org/10.1145/3691620.3695514>.
30. Adams, B. Co-evolution of source code and the build system. In Proceedings of the 2009 IEEE International Conference on Software Maintenance, Edmonton, AB, Canada, 20–26 September 2009; pp. 461–464. <https://doi.org/10.1109/ICSM.2009.5306272>.
31. McIntosh, S. Build system maintenance. In Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 21–28 May 2011; pp. 1167–1169. <https://doi.org/10.1145/1985793.1986031>.

32. Nejati, M.; Alfadel, M.; McIntosh, S. Code Review of Build System Specifications: Prevalence, Purposes, Patterns, and Perceptions. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 1213–1224. <https://doi.org/10.1109/ICSE48619.2023.00108>.
33. Xia, X.; Zhou, X.; Lo, D.; Zhao, X. An Empirical Study of Bugs in Software Build Systems. In Proceedings of the 2013 13th International Conference on Quality Software, Nanjing, China, 29–30 July 2013, pp. 200–203. <https://doi.org/10.1109/QSIC.2013.60>.
34. Besker, T.; Martini, A.; Bosch, J. Software developer productivity loss due to technical debt—A replication and extension study examining developers’ development work. *J. Syst. Softw.* **2019**, *156*, 41–61. <https://doi.org/https://doi.org/10.1016/j.jss.2019.06.004>.
35. Besker, T.; Martini, A.; Bosch, J. Technical Debt Cripples Software Developer Productivity: A Longitudinal Study on Developers’ Daily Software Development Work. In Proceedings of the 2018 IEEE/ACM International Conference on Technical Debt (TechDebt), Gothenburg, Sweden, 27–28 May 2018; pp. 105–114.
36. Maudoux, G.; Mens, K. Correct, Efficient, and Tailored: The Future of Build Systems. *IEEE Softw.* **2018**, *35*, 32–37. <https://doi.org/10.1109/MS.2018.111095025>.
37. Neitsch, A.; Wong, K.; Godfrey, M.W. Build system issues in multilanguage software. In Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 140–149. <https://doi.org/10.1109/ICSM.2012.6405265>.
38. Xiao, T.; Wang, D.; McIntosh, S.; Hata, H.; Kula, R.G.; Ishio, T.; Matsumoto, K. Characterizing and Mitigating Self-Admitted Technical Debt in Build Systems. *IEEE Trans. Softw. Eng.* **2022**, *48*, 4214–4228. <https://doi.org/10.1109/TSE.2021.3115772>.
39. Accellera SYSTEMS INITIATIVE. SystemRDL 2.0 Register Description Language. 2025. Available online: https://www.accellera.org/images/downloads/standards/systemrdl/SystemRDL_2.0_Jan2018.pdf (accessed on 15 March 2026).
40. Gasson, N. nvc—VHDL Compiler and Simulator. 2025. Available online: <https://github.com/nickg/nvc> (accessed on 15 March 2026).
41. Snyder, W. Verilator—Verilator Open-Source SystemVerilog Simulator and Lint System. 2025. Available online: <https://github.com/verilator/verilator> (accessed on 16 March 2026).
42. Kruszewski, M. HBS—Hardware Build System. 2025. Available online: <https://github.com/m-kru/hbs> (accessed on 16 March 2026).
43. Werner, M.; Zeraliu, I.; Han, Z.; Prebeck, S.; Servardei, L.; Ecker, W. Optimized HW/FW Generation from an Abstract Register Interface Model. In Proceedings of the 2020 23rd Euromicro Conference on Digital System Design (DSD), Virtual, 26–28 August 2020; pp. 35–39. <https://doi.org/10.1109/DSD51259.2020.00017>.
44. Cong, J.; Liu, B.; Neuendorffer, S.; Noguera, J.; Vissers, K.; Zhang, Z. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. -Comput.-Aided Des. Integr. Circuits Syst.* **2011**, *30*, 473–491. <https://doi.org/10.1109/TCAD.2011.2110592>.
45. Nane, R.; Sima, V.M.; Pilato, C.; Choi, J.; Fort, B.; Canis, A.; Chen, Y.T.; Hsiao, H.; Brown, S.; Ferrandi, F.; et al. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Trans. -Comput.-Aided Des. Integr. Circuits Syst.* **2016**, *35*, 1591–1604. <https://doi.org/10.1109/TCAD.2015.2513673>.
46. Kindgren, O. FuseSoc Generators: Produce and Specialize Cores on Demand. 2025. Available online: https://fusesoc.readthedocs.io/en/stable/user/build_system/generators.html (accessed on 4 March 2026).
47. Lahti, S.; Sjövall, P.; Vanne, J.; Hämäläinen, T.D. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Trans. -Comput.-Aided Des. Integr. Circuits Syst.* **2019**, *38*, 898–911. <https://doi.org/10.1109/TCAD.2018.2834439>.
48. Lahti, S.; Rintala, M.; Hämäläinen, T.D. Leveraging Modern C++ in High-Level Synthesis. *IEEE Trans. -Comput.-Aided Des. Integr. Circuits Syst.* **2023**, *42*, 1123–1132. <https://doi.org/10.1109/TCAD.2022.3193646>.
49. Hong, J.; Zhang, B.; Mao, G.; Hung, P.S.Y.; Cheung, R.C.C. Systematic HLS Co-Design: Achieving Scalable and Fully-Pipelined NTT Acceleration on FPGAs. *Electronics* **2025**, *14*, 3922. <https://doi.org/10.3390/electronics14193922>.
50. Zabolotny, W.M. Versatile DMA Engine for High-Energy Physics Data Acquisition Implemented with High-Level Synthesis. *Electronics* **2023**, *12*, 883. <https://doi.org/10.3390/electronics12040883>.
51. Graphviz. DOT Language. 2025. Available online: <https://graphviz.org/doc/info/lang.html> (accessed on 13 July 2026).
52. Kravchenko, V. Pipreqs. 2025. Available online: <https://github.com/bndr/pipreqs> (accessed on 2 March 2026).
53. Gasson, N. Nvc: A VHDL Simulator and Compiler. 2026. Available online: <https://www.nickgasson.net/nvc/> (accessed on 3 March 2026).
54. Kindgren, O. Edalize. 2025. Available online: <https://github.com/olofk/edalize> (accessed on 6 March 2026).
55. Gingold, T. GHDL: An Open-Source VHDL Simulator. 2026. Available online: <https://ghdl.github.io/ghdl/> (accessed on 3 March 2026).
56. Meyer, A.N.; Fritz, T.; Murphy, G.C.; Zimmermann, T. Software developers’ perceptions of productivity. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, 16–21 November 2014; FSE 2014; pp. 19–29. <https://doi.org/10.1145/2635868.2635892>.

57. Kindgren, O. FuseSoc Flags. 2025. Available online: https://fusesoc.readthedocs.io/en/stable/user/build_system/flags.html (accessed on 10 March 2026).
58. Snyder, W. Verilator: The Verilog HDL Simulator and Converter. 2026. Available online: <https://www.veripool.org/verilator/> (accessed on 3 March 2026).
59. Kazemitabaar, M.; Chow, J.; Ma, C.K.T.; Ericson, B.J.; Weintrop, D.; Wigdor, D. Studying the Effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. *ACM Trans. Comput. Educ.* **2024**, *1*, 1–23. <https://doi.org/10.1145/3544548.3580919>.
60. Thakur, S. Ahmad, B. Pearce, H. Tan, B. Dolan-Gavitt, B. Karri, R. and Garg, S. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Trans. Des. Autom. Electron. Syst.* **2024**, *29*, 31. <https://dl.acm.org/doi/10.1145/3643681>.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.