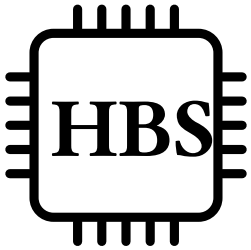

Hardware Build System
User Manual

Revision 0.0

19 December 2025

Abstract

This document serves as the official user manual for Hardware Build System (HBS). HBS is a Tcl-based, minimal common abstraction approach for build system for hardware designs. The main goals of the system include simplicity, readability, a minimal number of dependencies, and ease of integration with the existing Electronic Design Automation (EDA) tools.

keywords: automation, hardware build system, hardware design, hardware synthesis, project maintenance, testbench runner, simulation, FPGA, ASIC, productivity

Contents

Participants

Michał Kruszewski, *Chair*, *Technical Editor*, mkru@protonmail.com

Glossary

Not all terms defined in the glossary list are used in the user manual. Some of them are formally defined because they are helpful when discussing, for example, core definition.

core

Tcl namespace in which `hbs::Register` proc is called.

core name

Name of the Tcl namespace in which `hbs::Register` is called. For example, if `hbs::Register` is called in namespace `lib::pkg::my-core`, then `my-core` is the core name.

core path

Tcl namespace for the core. For example, if `hbs::Register` is called in namespace `lib::pkg::my-core`, then `lib::pkg::my-core` is the core path.

dependency

A target on which at least one other target depends. The dependency is an argument for at least one `hbs::AddDep` proc call.

depender

A target depending on at least one another target. Within a depender body the `hbs::AddDep` proc is called at least once.

flow

An ordered set of actions taken by a tool to produce a result specified by a user.

hbs file

A file with `.hbs` extension containing valid Tcl code.

proc

A Tcl procedure.

stage

A piece of a tool flow with a clearly defined task and output. The number and types of stages depend on a tool. For example, the GHDL has analysis, elaboration and simulation stages.

target

A proc, which name does not start with the floor character (_), defined in core.

target name

The name of the target in the target path. For example, if the target path is `lib::pkg::my-core::my-target`, then the target name is `my-target`.

target path

The Tcl path for the target. For example, if proc `my-target` is defined in the core with the core path `lib::pkg::my-core`, then the target path is `lib::pkg::my-core::my-target`.

tool

A software capable of processing hardware description sources or output from another tool. Example tools are: GHDL, Verilator, yosys, Vivado, nvc, etc.

to run a target

To execute commands defined in the target.

1 Overview

Hardware Build System (HBS) is a build system for hardware design projects. A build system for hardware design collects and processes files required for FPGA programming, ASIC production, running functional simulation, or carrying out formal verification. The files required to obtain desired output usually include much more than just classic hardware description files such as VHDL or SystemVerilog sources. For example, any synthesis or place and route tool requires also design contraints, at least for pin location and timing closure analysis. Moreover, most of real projects are not implement from scratch and utilize third-party IP cores. Those IP cores might be provided in different formats. Sometimes, they might even be encrypted. HBS tries to support all files that might potentially be required to generate final result. HBS does not limit to managing only pure hardware description files.

HBS was created out of frustration with all existing build systems for hardware designs.

2 Installation

All installation methods require that `hbs` and `hbs.tcl` files are placed in the same directory. There are four preferred installation methods.

1. Copy `hbs` and `hbs.tcl` files to your project. This is preferred if you want to modify HBS source files change its default behavior. It is not advised to change the default behavior, but if you need, feel free to adjust the build system to your project needs.
2. Copy `hbs` and `hbs.tcl` files to one of the directories in the `$PATH` environment variable.
3. Clone the repository and add its path to the `$PATH` environment variable.
4. Clone the repository and add an alias to the `hbs` file in `.bashrc` file (or equivalent).

2.1 Dependencies

HBS has three dependencies, one mandatory and two optional.

1. `tclsh (version >= 8.5)`.
2. `python3` - required for testbench target detection, automatic testbench running and dependency graph generation. If mentioned functionalites are not required, you can directly used `hbs.tcl` script instead of the `hbs` Python wrapper.
3. `graphviz` - required only if user wants to generate a dependency graph.

3 Internal architecture

It was decided that HBS will be implemented using Tcl language because of 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.
2. If the EDA tool provides the Tcl interface, then the Tcl shell is provided by the EDA tool vendor. The shell is nstalled during the installation of vendor tools. This implies that, in some cases, the build system user does not even have to install additional programs.
3. Executing arbitrary programs in arbitrary places in Tcl is very simple. 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 straighforward.

One of the most important things while designing the HBS was the separation of common build process operations that would constitute a common abstraction layer over EDA tools. At the end, it was decided that the following actions should constitute the common abstraction layer:

1. Target device setting - some EDA tools use the term “part” instead of “device,” for example, Vivado. Simulation EDA tools do not require a device setting. However, 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 can not handle analyzing different design units using different standard revisions, for example, nvc. In such a case, the build system must decide what the common standard revision shall be used for analyzing all HDL files.
5. Dependency specification - this is the core feature of any build system.
6. Generics/parameters setting - configuring parametric designs must be an inherent feature of any hardware build system.
7. Design top module setting - all EDA tools carrying out simulation or synthesis requires information on the top module.
8. Code generation - a hardware build system must provide a simple way for automatic arbitrary code generation. This is further explained in Section 3.12.

3.1 General structure

The HBS is implemented in two files `hbs.tcl` and `hbs`. The first is implemented in Tcl, and the second in Python. The `hbs.tcl` file implements all the core features related directly to the interaction with the EDA tools. The `hbs.tcl` provides the following functions:
1. Dumping information about detected cores in JSON format.
2. Listing cores found in hbs files.
3. Listing targets for a given core.
4. Running target provided as a command line argument.

The `hbs` is a wrapper for the `hbs.tcl` and serves the following additional functions:
1. Showing documentation for hbs Tcl symbols.
2. Generating dependency graph.
3. Listing testbench targets.
4. Running testbench targets.

By default, the user calls the `hbs` program. However, if none of the additional functions are required, the user can call the `hbs.tcl` directly. In such a case, the whole build system is limited to a single file.

3.2 Cores and cores detection

When the user calls `hbs` (or `hbs.tcl`), all directories, starting from the working directory, are recursively scanned to discover all files with the `.hbs` extension (symbolic links are also scanned). Files with the `.hbs` extension are regular Tcl files that are sourced by the `hbs.tcl` script. However, before sourcing hbs files, the file list is sorted so that scripts with shorter path depth are sourced as the first ones. 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 controlling 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

separate `utils.hbs` file contaning utility procedures, and place it in the the project root directory. This guarantees that `utils.hbs` will be sourced before any hbs file in subdirectories. Within hbs files, the user usually defines cores and targets, although the user is free to have any valid Tcl code in hbs files.

The below snippet 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.

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

To register a core, the user must explicitly call `hbs::Register` procedure at the end of the core namespace. Such a mechanism helps to distinguish regular Tcl namespaces from Tcl namespaces representing core definitions. If the user forgets to register a core, the build system gives a potential hint. An example error message is presented below.

```
[user@host tmp] hbs run lib::core::tb
checkTargetExists: core 'lib::core' not found, maybe the core is not:
  1. registered 'hbs doc hbs::Register',
  2. sourced 'hbs doc hbs::IgnoreRegexes'.
```

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 the below snippet. In this case, the flip-flop core path is `vendor::library::flip-flop::1.0`.

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

2. The user can define arbitrary deep core paths (limited by the Tcl shell). This is presented in the below snippet. In this case, the core path consists of seven parts.

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

3. The user can nest namespaces to imitate the structure of libraries and packages. This is presented in Listing 6. Three flip-flop cores are defined in the snippet. Listing 7 presents output for listing flip-flop cores.

If you every tried to use `tclsh` to REPL (read-eval-print-loop), you probably realized that `tclsh` by default does not support arrow keys. You can't fix a typo in a line without deleting some line content. There is also no command history support. However, this can be improved. The first solution is install the `rlwrap` programm and call `rlwrap tclsh` instead of `tclsh`. To make it shorter to type, you can define an alias for your shell of choice, for example, for bash alias `tclsh='rlwrap tclsh'`. The second option is to install the Tcl `tclreadline` package. This package often comes as OS distro package. For example, on Debian/Ubuntu, you can install it with `apt install tcl-tclreadline`. Once installed, create `.tclshrc` file in your home directory and add the following content:

```
package require tclreadline
tclreadline::Loop
```

Not only you will get support for arrow keys and command history, but also improved prompt.

5.2 Passing variadic arguments to proc

To pass variadic arguments to a proc, the last proc parameter must be called `args`. You can then easily iterate over the arguments using the `foreach` loop. The below example is taken directly from the hbs Tcl source code.

```
proc AddIgnoreRegex {args} {
  foreach reg $args {
    hbs::dbg "adding ignore regex $reg"
    lappend hbs::IgnoreRegexes $reg
  }
}
```