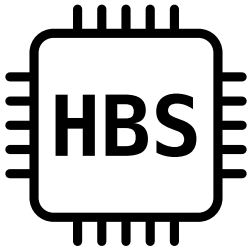


Hardware Build System
User Manual

Revision 1.0

11 January 2026

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 seamless integration with 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.

API

Application Programming Interface

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.

EDA

Electronic Design Automation

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.

run core

The core whose target is currently being run.

run target

The target which is currently being run.

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 context

An invariant environment constituting of HBS variables that are guaranteed to not implicitly change during the target proc evaluation.

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 execute a target

To execute the target procedure code by explicitly or implicitly calling the target procedure, for example, by calling `hbs::AddDep`.

to run a target

To call the `run` or `dry-run` HBS command for a given 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 the 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 design constraints, at least for pin location and timing closure analysis. Moreover, most real projects are not implemented 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 a final result. HBS is not limited to managing only pure hardware description files.

HBS can be described as a Tcl-based, minimal common abstraction approach for build system for hardware designs. This is because HBS implements a common abstraction layer that is:

- minimal,
- limited to the primary common features,
- implemented in Tcl.

However, do not be misled by the word minimal. Only the common abstraction layer is minimal. HBS is designed in such a way that it is straightforward to utilize EDA tools' exclusive features. This was achieved by implementing HBS in Tcl. The HBS build system code is executed directly by the Tcl shell embedded in EDA tools. This, in turn, grants direct access to the EDA tools' Tcl commands during the build execution. HBS is not a tool for preparing Tcl code that is later executed by EDA tools. HBS is a tool which code is executed by EDA tools.

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 HBS 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. 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, hence it should be hidden under the build system abstraction.
9. Code generation - a hardware build system must provide a simple way for automatic arbitrary code generation. This is further explained in Section 3.13.

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 Tcl naming conventions

Understanding Tcl naming conventions is crucial for using or contributing to the HBS. All HBS code is hidden under the `hbs` namespace. Code related to a particular tool is further hidden in the `hbs::{tool}` namespace.

Tcl does not allow defining private symbols within namespaces; all symbols are public. However, `hbs.tcl` differentiates between public and private symbols. Public symbols start with an uppercase letter, and private symbols begin with a lowercase letter.

The user should only use public symbols within hbs files. Although using private symbols is discouraged, it is not forbidden, and if you really know what you do, feel free to use them.

The `hbs` namespace consists of variables and procs. Even though some variables are public, the user shall not set them directly. They are public because they can be safely read from the hbs files. However, setting them might require some additional actions. For example, `hbs::Tool` is a public variable, but the user shall use `hbs::SetTool` function for setting the tool. There is no such requirement for getting the value of a public variable.

All variables representing choices (enumeration) use lowercase strings. For example, the `hbs::Tool` can be `"ghdl"`, `"vivado-prj"`, etc. The `hbs::ToolType` can equal `"formal"`, `"simulation"`, or `"synthesis"`. The point of this is to avoid error cases when one core maintainer sets the tool to GHDL, but another core maintainer has, for example, the following condition in one of the targets:

```
if {$hbs::Tool eq "ghdl"}
```

The expression would evaluate to false, although the tool is GHDL. Most `hbs::Set*` procedures assert that users provide lowercase names.

3.3 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 following snippet presents a very basic flip-flop core definition:

```
namespace eval flip-flop {
  proc src {} {
```

```
    hbs::AddFile flip-flop.vhd
  }
  hbs::Register
}
```

The flip-flop core has a single target named `src`. The core consists of a single VHDL file.

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 in the following snippet:

```
[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::FileIgnoreRegexes'.
```

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 following 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 following 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 following snippet.

   ```
   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
    }
  }
}
```

Three flip-flop cores are defined in the snippet. The following snippet presents output for listing flip-flop cores:

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

3.3.1 Excluding hbs files from being sourced

Sometimes a file with the `.hbs` extension is not a valid hbs file, or maybe you want to temporarily disable valid hbs files from being sourced. HBS provides a built-in mechanism for excluding files with the `.hbs` extension from being sourced. This is achieved using the `hbs::AddFileIgnoreRegex` function. You just have to call this function in one of valid hbs files. The function will be executed once the file containing the call is sourced.

Usually the hbs file containing calls to the `hbs::AddFileIgnoreRegex` proc is placed in the project root directory. This is becuase hbs files placed in the project root directory are sourced before hbs files placed in subdirectories. Order of hbs files sourcing is described in Section 3.3.

Arguments provided to the `hbs::AddFileIgnoreRegex` proc are treated as regular expressions. This allows for ignoring multiple paths using a single regex. However, you are free to provide multiple ignore regex, and all of them will be checked while sourcing hbs files.

3.3.2 Explicitly sourcing hbs files

Sometimes there might be a need to explicitly source an hbs file. For example, when you generate core code and would like also to generate the hbs file for the core. HBS automatically searches for hbs files only when `hbs.tcl` file starts running. If you generate code within hbs files, the newly generated hbs file will not be automatically discovered. However, you can easily source the newly generated hbs file. Simply use the Tcl built-in `source` command.

3.4 Targets and targets detection

HBS automatically detects targets. Targets are all Tcl procedures defined in the scope of core namespaces (namespaces with a call to `hbs::Register`). 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. The following snippet presents an 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
}
```

The core path is `vhdl::simple::edge-detector`. The core has three targets: `src`, `tb-sync`, `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 testbench targets.

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

3.5 Testbench targets

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

1. start with the `tb-` or `tb_` prefix,
2. end with the `-tb` or `_tb` suffix,
3. equal `tb`.

For example, for the following hbs file:

```
namespace eval my-core {
  proc tb {} {
    puts "Hello from tb"
  }
  proc my-tb {} {
    puts "Hello from my-tb"
  }
  proc tb_my {} {
    puts "Hello from tb_my"
  }
  hbs::Register
}
```

The `hbs` program detects the following testbench targets:

```
[user@host tmp] hbs list-tb
my-core::my-tb
my-core::tb
my-core::tb_my
```

3.6 Running targets

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 the following snippet:

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

Results with the following output:

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

However, in most cases, you want to run a target related to the flow of the set EDA tool. In such a case, instead of manually calling all of the required tool commands, you can call the `hbs::Run` procedure in the core target procedure. The `hbs::Run` procedure has an optional argument accepting the stage after which the tool flow should stop. This is further described in Section 3.10. After `hbs::Run` returns, the user can continue processing. For example, the user can run scripts analyzing code coverage or preparing additional reports.

3.7 Target parameters

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

```
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. The blow snippet presents output from running the target with different `stage` parameter values.

```
[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 testbench target from the command line or changing the top-level module. What target parameters are used for is limited only by your imagination, and Tcl semantics.

3.8 Target context

As an engineer implementing a given core, you control the dependencies of the core. However, you do not control who will use the core and how. As targets are regular Tcl procedures, there is a need for a mechanism allowing the core author to evaluate the target procedure in the invariant environment. Such a mechanism in HBS is called the target context.

The target context assures that the following variables are not affected by dependee or dependency target execution:

1. HDL library,
2. HDL standard,
3. top module name,
4. arguments prefix,
5. arguments suffix,
6. the core path,
7. the core name,
8. the target path,
9. the target name.

The following snippet presents an example of the target context mechanism:

```
namespace eval pkg {
  namespace eval foo {
    proc src-foo {} {
      hbs::SetLib "lib-foo"
      hbs::AddDep pkg::bar::src-bar
      puts "foo lib: $hbs::Lib"
      puts "foo core: $hbs::ThisCorePath"
      puts "foo target: $hbs::ThisTarget"
    }
    hbs::Register
  }
  namespace eval bar {
    proc src-bar {} {
      hbs::SetLib "lib-bar"
      puts "bar lib: $hbs::Lib"
      puts "bar core: $hbs::ThisCorePath"
      puts "bar target: $hbs::ThisTarget"
    }
    hbs::Register
  }
}
```

The following snippet presents the output from running the `pkg::foo:src-foo` target:

```
[user@host tmp] hbs run pkg::foo::src-foo
bar lib: lib-bar
bar core: pkg::bar
bar target: src-bar
foo lib: lib-foo
foo core: pkg::foo
foo target: src-foo
```

As can be seen, setting library in a target of one core, does not affect library in the target of another core.

3.9 Target dependencies

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

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

To add multiple distinct dependencies, the user must call `hbs::AddDep` multiple times. The ability to pass custom arguments to 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 so that generating a dependency graph is possible. 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 some target procedure rerun, the user can always directly call the target. However, enforcing target procedure rerun usually is an alert that a regular Tcl procedure shall be used instead of the core target procedure.

The bloew snippet contains an example core definitions for presenting target dependency rules.

```
namespace eval core-a {
  proc target {} {
    hbs::AddDep core-b::target
    hbs::AddDep core-c::target
    hbs::AddDep generator-core::gen a
    hbs::AddDep generator-core::gen x
    puts "core-a::target"
  }
  hbs::Register
}
namespace eval core-b {
  proc target {} {
    hbs::AddDep core-c::target
    hbs::AddDep generator-core::gen b
    hbs::AddDep generator-core::gen x
    puts "core-b::target"
  }
  hbs::Register
}
namespace eval core-c {
  proc target {} {
    puts "core-c::target"
```

```
  }
  hbs::Register
}
namespace eval generator-core {
  proc gen {arg} {
    puts "generator-core::gen $arg"
  }
  hbs::Register
}
```

There are four cores: `core-a`, `core-b`, `core-c`, `generator-core`. `core-a` depends on `core-b` and `core-c`. `core-b` depends on `core-c`. Moreover, cores `core-a` and `core-b` depend on the `generator-core`. However, they use different argument values for generation.

The following snippet presents output from running target `core-a::target`:

```
[user@host tmp]$ hbs run core-a::target
core-c::target
generator-core::gen b
generator-core::gen x
core-b::target
generator-core::gen a
core-a::target
```

As can be seen, the `core-c::target` is run only once, even though both `core-a::target` and `core-b::target` depend on it. This is because `core-c::target` has no arguments and can be added as a dependency only once. On the other hand, `generator-core::gen` is run three times. This is because `generator-core::gen` is added as a dependency four times, and three times with a distinct argument value.

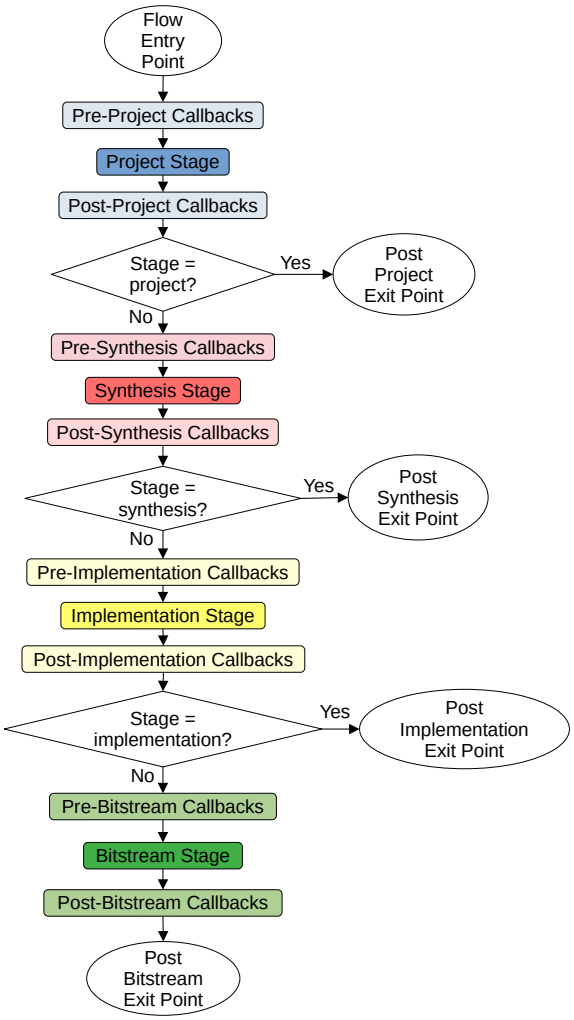
3.10 EDA tool flow and stages

The primary function of any hardware build system is to provide the ability to build a design. What the term “build” actually means usually depends on the EDA tool. For some, it is only synthesis; for others, it is synthesis and implementation, and yet for others it might be simulation or just code linting. Each EDA tool is characterized by a distinct flow consisting of different stages. To mimic this behavior, HBS supports the following stages (alphabetical order):

1. analysis - HDL files analysis,
2. bitstream - device bitstream generation,
3. elaboration - design elaboration,
4. implementation - design implementation,
5. project - project creation,
6. simulation - design simulation,
7. synthesis - design synthesis.

Not all stages make sense for all EDA tools. Which stages are present in the given tool flow depends on that tool implementation. Namely, on the implementation of the `hbs::<tool>::run` procedure. This is a tool private procedure. This procedure is called at the end of the `hbs::Run` procedure execution, which is a HBS public procedure.

The following figure shows the structure of the tool flow for the Vivado project mode (`vivado-prj`):



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

The post stage callbacks and pre stage callbacks for adjacent stages were added for two reasons.

1. To clearly communicate which stage the callback refers to. For example, configuring implementation settings based on the synthesis results can be done in a post-synthesis callback or pre-implementation callback. However, as the callback modifies the implementation settings, it is probably better to add it using the `hbs::AddPreImplCb`, than `hbs::AddPostSynthCb`.
2. To introduce, to some extent, a manageable order of callbacks execution. The pre and post callbacks of a given stage are executed in the order they are added. If some nested dependencies add callbacks for the same pre or post stage, then the order of callbacks execution depends on the order of `hbs::AddDep` calls. However, if the result of one of the callbacks depends on the result of the other one, then relying on a user to call the `hbs::AddDep` procedures in proper order is error prone. In such a case, the callback that must be executed as the first one can be added to the post-synthesis callbacks, and the second callback can be added to the pre-implementation callbacks. Such an approach is immune to the order of `hbs::AddDep` calls.

You can utilize stage callbacks in any desired way. However, the primary purpose of stage callbacks is to adjust the design build based on the results from a particular stage. For example, you might want to configure additional implementation settings based on the synthesis results. You might even terminate the tool flow in a given callback and report an error if certain conditions are not met.

To get to know stages supported by a given EDA tool you can call `'hbs doc <tool>'` command. The following snippet presents documentation message for the GHDL simulator.

```
[user@host tmp] hbs doc ghdl
# GHDL simulator
#
# HBS requires that GHDL is compiled with the LLVM or GCC backend.
# It does not support GHDL with mcode backend.
#
# GHDL supports the following stages:
#   - analysis,
#   - elaboration,
#   - simulation.
```

3.11 EDA tool commands custom arguments

The HBS has a minimal common abstraction layer. You perform all the actions not covered by the abstraction layer by directly calling EDA tool commands. For example, generating extra timing or clock network reports. However, some of the actions are hidden under the HBS API. For example, adding or analyzing HDL files. EDA tool commands used to perform actions hidden under common API usually have some parameters that are not used by default. Nevertheless, sometimes there is a need to specify additional parameters. In such a case, there are two possible solutions.

1. The first option is to bypass the HBS API and directly call the underlying EDA tool command. The drawback of this approach is that the user must manually handle the current context. For example, when adding an HDL file, the user must manually specify the library or standard revision. Bypassing HBS API also bypasses the target context!
2. The second option is to set the underlying command arguments prefix or suffix. This can be achieved with the `hbs::SetArgsPrefix` and `hbs::SetArgsSuffix` procedures. The argument prefix is always appended after the command name, and the argument suffix is always appended after all command arguments.

The following snippet presents Ethernet Management Data Input/Output (MDIO) core definition:

```
namespace eval vhdl::ethernet {
  namespace eval mdio {
    proc src {} {
      hbs::SetLib "ethernet"
      hbs::AddFile mdio.vhd
    }
    proc tb {} {
      hbs::SetTool "nvc"
      hbs::AddPostElabCb hbs::SetArgsPrefix "--messages=compact"
      src

      hbs::SetLib ""
      hbs::AddFile tb/tb-mdio.vhd
      hbs::SetTop "tb_mdio"
      hbs::Run
```

```
    }
    hbs::Register
  }
}
```

The core has one testbench target utilizing the nvc simulator. Nvc report messages occupy multiple lines by default. However, this can be changed by specifying the `--messages=compact` argument when running the simulation. As running the simulation is the last stage of the nvc flow, the call to `hbs::SetArgsPrefix` must be wrapped by the call to the `hbs::AddPostElabCb` procedure.

The following snippet shows commands executed by the HBS to run the simulation:

```
[user@host vhdl-ethernet]$ hbs run vhdl::ethernet::mdio::tb
nvc --std=2019 -L. --work=ethernet -a vhdl/vhdl-ethernet/mdio.vhd
nvc --std=2019 -L. --work=work -a vhdl/vhdl-ethernet/tb/tb-mdio.vhd
nvc --std=2019 -L. -e tb_mdio
nvc --messages=compact --std=2019 -L. -r tb_mdio --wave
```

The `--messages=compact` argument was appended right after the `nvc` command.

3.12 HBS API extra symbols

The HBS API consists not only of symbols related to the common EDA abstraction layer. For example, there are extra `hbs::Exec` and `hbs::CoreDir` procedures. The first one is a wrapper for the Tcl standard `exec` procedure. Before calling `exec`, the `hbs::Exec` changes the working directory to the directory where the currently evaluated core is defined. When `exec` returns, the `hbs::Exec` restores the working directory. The `hbs::CoreDir` procedure allows the user to get the path of the directory in which the currently evaluated core is defined.

HBS also provides users with the following extra variables:

1. `hbs::ThisCorePath` - the path of the core which target is currently being run,
2. `hbs::ThisCoreName` - the name of the core which target is currently being run,
3. `hbs::ThisTargetPath` - the path of the target which is currently being run,
4. `hbs::ThisTargetName` - the name of the target which is currently being run,
5. `hbs::RunCorePath` - the path of the run core,
6. `hbs::RunCoreName` - the name of the run core,
7. `hbs::RunTargetPath` - the path of the run target,
8. `hbs::RunTargetName` - the name of the run target,
9. `hbs::RunTargetArgs` - the list with command line arguments passed to the run target.

To get the list of all HBS public symbols you can run `'hbs doc'` command in the shell.

3.13 Code generation

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

Some existing hardware build systems do not allow the direct calling of an arbitrary external program in arbitrary places during the build process. Instead, you have to define so-called generators. Only then can you call the generator wihtin 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 they are used, which decreases 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 shell seems natural.

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

The bellow snippet presents an example of calling an external code generator tracked by the dependnecy 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
}
```

In actual usage, the call to the shell `echo` command would be replaced with a call to the proper code generator program. Calls to the `hbs::AddFile` are commented out because no EDA tool was set.

The following snippet presents how to achieve the same result without tracking the generator as a dependency.

```
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
}
```

This task is even more straightforward, as you can call an external generator program directly in the target procedure.

3.14 HDL default standard revisions

There are numerous discrepancies between EDA tools in supporting various HDL standard revisions. For example:

- the default standard revision might differ,
- the minimum supported standard revision might differ,
- the maximum supported standard revision might differ,
- the set of supported standard revisions might vary.

HBS tries to unify the default HDL standard revision in the least disruptive way.

For VHDL, the default standard revision is 2008. The 2008 revision was a significant language modernization with numerous useful features being added. Moreover, it was widely adopted and is supported in all actively maintained EDA tools.

For Verilog/SystemVerilog, the default standard revision is 2012. This revision probably has the widest adoption. Moreover, UVM was designed to work with revision 2012. Some tools come with the standard revision 2017 being the default one. However, most of them support only part of the things introduced in this revision.

When you set standard revision (`hbs::SetStd` procedure) in your hbs files, it might turn out that a particular tool does not support this standard, even if the set revision is a valid revision for a given language. Two things might happen in this scenario.

1. If the tool does not support the standard revision you set and any higher standard revision, `hbs.tcl` will exit with an error. An example error message is presented in the following snippet:

   ```
   core::target: hbs::ghdl::addVhdlFile: /home/user/workspace/hbs-tests/SetStd/
   ghdl-unsupported-std/abc.vhd: ghdl doesn't support VHDL standard '2019'
   ```

2. If the tool supports any higher standard revision, the set standard revision will be automatically upgraded to the nearest standard supporting features present in the standard you set. For example, you cannot enforce compatibility with SystemVerilog standard revisions 2005, 2009, and 2012 in Gowin. However, Gowin claims support for SystemVerilog standard revision 2017. In such a case, if you set the standard to 2005, 2009, or 2012, it will be automatically upgraded to 2017.

3.15 HBS environment variables

HBS utilizes some environment variables. Some of them are used internally, for example, `HBS_TOOL_BOOTSTRAP`, and some can be set by the user to control HBS behavior.

3.15.1 HBS_TOOL_BOOTSTRAP - hbs.tcl bootstraps itself

The `HBS_TOOL_BOOTSTRAP` environment variable is entirely managed by the `hbs.tcl` file. Do not set or unset this variable manually. The variable is required because `hbs.tcl` sometimes must bootstrap itself. For example, if `hbs::Tool` is set to `"vivado-prj"`, but `hbs.tcl` was run with the OS Tcl shell (`tclsh`), then `hbs.tcl` must bootstrap itself with the Tcl shell embedded in the Vivado.

3.15.2 HBS_DEBUG - debugging build flow

By setting the `HBS_DEBUG` environment variable, you can enable debug messages. Debug messages are printed to the standard error. This means that you can enable debug messages in dry runs and still be able to redirect debug messages and tool commands independently.

You can extend debug messages with custom messages from your hbs files. There is `hbs::Debug` procedure which you can use for conditionally printing messages when the `HBS_DEBUG` environment variable is set. Messages printed with the `hbs::Debug` are always prefixed with the path of the procedure in which `hbs::Debug` is called. See `'hbs doc Debug'` for more details.

If you need a more advanced logging mechanism with multiple levels, you can implement it on top of `hbs::Debug`. Alternatively, you can implement a custom logging mechanism. To check if the `HBS_DEBUG` environment variable is set you can check the value of the `hbs::DebugEnvSet` variable instead of

calling `[info exists ::env(HBS_DEBUG)]`. HBS is not planned to natively support multi-level logging mechanism. The dry runs and `HBS_DEBUG` are probably more than enough for debugging build flows.

3.15.3 HBS_DEVICE - enforcing device

By setting the `HBS_DEVICE` environment variable, you can enforce the value of the `hbs::Device`. If `HBS_DEVICE` is set, then `hbs.tcl` during initialization (before any hbs file is sourced) sets the value of `hbs::Device` to the value of `HBS_DEVICE`. If `HBS_DEVICE` is set, any call to the `hbs::SetDevice` is ignored.

The `HBS_DEVICE` variable might be useful for determining the target device for the build from the shell. Similar functionality can be achieved using the target parameters described in Section 3.7. However, you may want to utilize target parameters for different purposes.

3.15.4 HBS_EXIT_SEVERITY - enforcing exit severity

By setting the `HBS_EXIT_SEVERITY` environment variable, you can enforce the value of the `hbs::ExitSeverity`. If `HBS_EXIT_SEVERITY` is set, then `hbs.tcl` during initialization (before any hbs file is sourced) sets the value of `hbs::ExitSeverity` to the value of `HBS_EXIT_SEVERITY`. If `HBS_EXIT_SEVERITY` is set, any call to the `hbs::ExitSeverity` is ignored.

The `HBS_EXIT_SEVERITY` environment variable is useful for quickly running testbenches with modified exit severity. For example, your simulation suddenly starts failing, and you would like to stop it when the first warning is encountered.

3.15.5 HBS_TOOL - enforcing tool

By setting the `HBS_TOOL` environment variable, you can enforce the value of the `hbs::Tool`. If `HBS_TOOL` is set, then `hbs.tcl` during initialization (before any hbs file is sourced) sets the value of `hbs::Tool` to the value of `HBS_TOOL`. If `HBS_TOOL` is set, any call to the `hbs::SetTool` is ignored.

The `HBS_TOOL` environment variable is helpful in running testbench targets with different simulators. If you want to run just a single testbench target with multiple simulators, then you can use a target parameter (see Section 3.7) for your testbench target, or you can set the `HBS_TOOL` environment variable. However, if you want to run multiple testbench targets using the `'hbs test'` command, you must utilize the `HBS_TOOL` environment variable. This is because `'hbs test'` does not support passing arguments to testbench targets.

Be careful! Some simulators, for example, `nvc` and `questa`, might share some directories or file names. If you run a testbench with one simulator, then running the same testbench with a different simulator in the same directory without cleaning it might result in errors. It is advised to clean the build directory before running the same testbench with a different simulator. If you run your testbenches with multiple simulators in the continuous integration pipeline, then you probably want to store all build results for all simulators. In such a case, you can change the build directory for other simulators using the `HBS_BUILD_DIR` environment variable. This is presented in the following snippet:

```
# Run all testbenches with nvc and place them in build-nvc directory.
export HBS_TOOL=nvc
export HBS_BUILD_DIR=build-nvc
hbs test
# Run the same testbenches with questa and place them in build-questa directory.
export HBS_TOOL=questa
export HBS_BUILD_DIR=build-questa
hbs test
```

3.15.6 HBS_STD - enforcing HDL standard revision

By setting the `HBS_STD` environment variable, you can enforce the value of the `hbs::Std`. If `HBS_STD` is set, then `hbs.tcl` during initialization (before any hbs file is sourced) sets the value of `hbs::Std` to the value of `HBS_STD`. If `HBS_STD` is set, any call to the `hbs::SetStd` is ignored.

The `HBS_STD` environment variable is analogous to the `HBS_TOOL` environment variable. However, running multiple, or even one, testbench targets with different HDL standard revisions is probably not useful. The `HBS_STD` environment variable is rather handy for quickly checking if a given target can run with a different HDL standard revision.

3.15.7 HBS_BUILD_DIR - changing build directory

By setting the `HBS_BUILD_DIR` environment variable, you can enforce the value of the `hbs::BuildDir`. If `HBS_BUILD_DIR` is set, then `hbs.tcl` during initialization (before any hbs file is sourced) sets the value of `hbs::BuildDir` to the value of `HBS_BUILD_DIR`. If `HBS_BUILD_DIR` is set, any call to the `hbs::SetBuildDir` is ignored. The usefulness and functionality of `HBS_BUILD_DIR` are described in Section 3.15.5

4 Command line interface commands

4.1 `doc` - viewing HBS API documentation

The `doc` command was added to ease viewing documentation for HBS Tcl symbols. The command is executed by the `hbs` file, so Python is required for the command to work. If no argument is provided for the `doc` command, then `hbs` prints a list of all HBS Tcl public symbols. To get more information on the particular symbol, simply provide it as an argument for the `doc` command. The following snippet presents an example of `doc` command output:

```
[user@host ~] hbs doc SetStd
# Sets standard revision for HDL files.
#
# To get the value of currently set standard revision simply
# read the value of hbs::Std variable.
#
# Standard revision for a given file must be set before adding a file.
# For example:
#   hbs::SetStd 2008
#   hbs::AddFile entity.vhd
proc SetStd {std}
```

4.2 `dump-cores` - dumping cores information

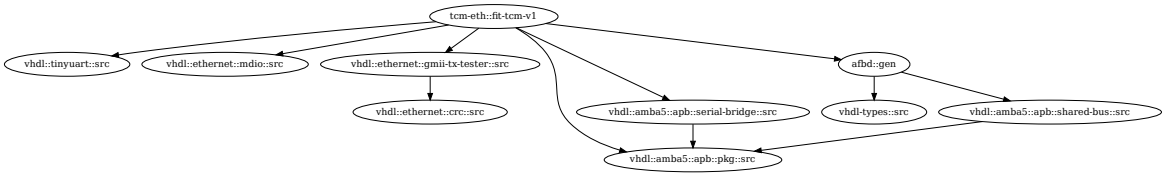
The `dump-cores` command allows dumping information about found cores into JSON format. The generated JSON data can be used for further processing. For example, the graph command utilizes data from JSON dump to generate the dependency graph. The command is executed by the `hbs.tcl` file, so Python is not required for the command to work. If you do not like the default behavior of the `hbs` Python wrapper, you can write your own. Simply utilize dumped JSON data as a stream from `hbs.tcl` to your wrapper.

4.3 `graph` - generating target dependency graph

The `graph` command allows generating target dependency graphs in PDF format. The command is executed by the `hbs` file, so Python is required for the command to work. Moreover, you must have `graphviz` installed on your machine.

The `graph` command requires information about cores in the JSON format. This implies that the user must execute the `dump-cores`, `run` or `dry-run` command before generating a dependency graph. However, this is not a major issue in practice. Dumping cores, even for large designs, does not take more than a few seconds, as the `dump-cores` command does not run the target tool flow. The `dry-run` command is also very fast, as it does not execute or evaluate any commands.

The following figure presents an example dependency graph generated for VSC8211 (Ethernet PHY) chip tester design:



4.4 `help` - displaying help message for commands

The `help` command serves as a standard help message display command. If no argument is provided, then the help message regards the `hbs` general use. If argument is provided for the `help` command, then it must be a valid command name. In such a case, `hbs` prints help message for the provided command. The following snippet presents help message for the `dump-cores` command:

```
[user@host tmp] hbs help dump-cores
Dump info about cores found in .hbs files in JSON format.

  hbs dump-cores

The JSON is directed to stdout.
If you want to save it in a file simply redirect stdout.
```

4.5 `list-cores` - listing cores found in hbs files

The `list-cores` command allows listing all cores discovered by the HBS. The `list-cores` command is executed by the `hbs.tcl` file, so the command does not require Python to work. The following snippet presents an output for listing all cores in the VHDL APB library:

```
[user@ahost apb] hbs list-cores
vhdl::amba5::apb::bfm
vhdl::amba5::apb::cdc-bridge
vhdl::amba5::apb::checker
vhdl::amba5::apb::crossbar
vhdl::amba5::apb::mock-completer
vhdl::amba5::apb::pkg
vhdl::amba5::apb::serial-bridge
vhdl::amba5::apb::shared-bus
```

The following snippet presents an output for listing various bridge cores in the same APB library:

```
[user@ahost apb] hbs list-cores bridge
vhdl::amba5::apb::cdc-bridge
vhdl::amba5::apb::serial-bridge
```

Please note that you can provided arbitrary strings to the `list-cores` command. The core is listed if its core path contains at least one string provided in arguments. For example, the following snippet presents an output for listing all cores containg the `bri` or `bar` string in the same APB library:

```
[user@host apb] hbs list-cores bri bar
vhdl::amba5::apb::cdc-bridge
vhdl::amba5::apb::crossbar
vhdl::amba5::apb::serial-bridge
```

4.6 `list-targets` - listing targets for discovered cores

The `list-targets` command allows listing all targets discovered by the HBS. The command is analogous to the `list-cores` command but works on targets instead of cores. The `list-targets` command is executed by the `hbs.tcl` file, so the command does not require Python to work. The following snippet presents an output for listing `src` targets in the VHDL APB library:

```
[user@host apb] hbs list-targets src
vhdl::amba5::apb::bfm::src
vhdl::amba5::apb::cdc-bridge::src
vhdl::amba5::apb::checker::src
vhdl::amba5::apb::crossbar::src
vhdl::amba5::apb::mock-completer::src
vhdl::amba5::apb::pkg::src
vhdl::amba5::apb::serial-bridge::src
vhdl::amba5::apb::shared-bus::src
```

The name "src" is preferred name for a core target if the core has only one target containing all sources required for core utilization. However, this is not a formal requirement, so feel free to name your targets however you want.

4.7 `list-tb` - listing testbench targets

The `list-tb` command allows listing all testbench targets discovered by HBS. The `list-tb` command is analogous to the `list-targets` command, but it works solely on testbench targets instead of all targets. The command is executed by the `hbs` file and requires Python to work. The following snippet presents an output for listing testbench targets for bridges in the VHDL APB library:

```
[user@host apb] hbs list-tb bridge
vhdl::amba5::apb::cdc-bridge::tb-to-faster
vhdl::amba5::apb::cdc-bridge::tb-similar-slower
vhdl::amba5::apb::cdc-bridge::tb-similar-faster
vhdl::amba5::apb::cdc-bridge::tb-to-slower
vhdl::amba5::apb::serial-bridge::tb-write
vhdl::amba5::apb::serial-bridge::tb-read
```

If no arguments are provided for the `list-tb` command, then all testbench targets for all discovered cores are listed.

4.8 `run` - running targets

The `run` command allows running target procedures. Usually, targets are run to carry out the build process or simulation. However, the user is free to carry out any action in the target being run. You can, for example, use targets for software recompilation.

Running targets is described in Section 3.6, Section 3.7, and Section 3.8.

4.9 `dry-run` - running target without executing and evaluating commands

The `dry-run` command runs a given target without executing and evaluating commands. It only prints the commands to the standard output for previewing the actions carried out by the target.

In the dry run, the following things change compared to the actual run:

1. `hbs.tcl` does not bootstrap itself with a proper Tcl shell from the EDA tool.
2. Shell, or EDA Tcl commands, are not executed or evaluated. They are only printed to the standard output.

The `dry-run` command is useful in the following scenarios:

1. When you want to generate a dependency graph without running any tool flow. Running targets including synthesis, or place and route, stages might be time-consuming. A dry run allows for quickly dumping cores into `.json` file for further dependency graph generation.
2. When you debug your hbs files or the build flow. The `dry-run` command allows for a quick preview of all commands that are executed or evaluated during the actual run.
3. When you want to generate a Tcl or shell script for building a project or running a simulation. Executing the `dry-run` command is like taking a snapshot of your build procedure. For example, you think you have found a bug in a simulator, and you would like to open an issue on GitHub. The project maintainer requires you to provide an example for reproducing the bug. However, the bug reproducing requires multiple files and shell commands to be executed. You can't expect the simulator developer to utilize HBS as a build system. To deliver shell commands for bug reproducing, you can simply copy the `dry-run` output.
4. For implementing HBS internal regression tests.

4.9.1 Supporting dry runs in hbs files

All the `hbs.tcl` internal code supports dry runs by default. However, if you want your hbs files to also support dry runs, you must obey some extra rules. You cannot directly call EDA Tcl custom commands. This is because `hbs.tcl` does not bootstrap itself with a proper Tcl shell from the EDA tool in the dry run. For example, let's assume you build a project using Vivado, and you have the following command in your hbs file:

```
set_msg_config -suppress -id "Synth 8-6014" -string {{REPORT_PREFIX}}
```

The command will simply fail during the dry run with the following message:

```
invalid command name "set_msg_config"
    while executing
...
```

Your OS Tcl shell (`tclsh`) does not have the built-in `set_msg_config` command. This is a Vivado custom command.

HBS provides three procedures supporting implementing dry run compatible user hbs files, the `hbs::Eval`, `hbs::Exec`, and `hbs::ExecInCoreDir`. The `hbs::Eval` procedure prints the command to the standard output and evaluates it only if the current run is not a dry run. All you have to do is to prepend EDA tool custom command with a call to the `hbs::Eval` procedure and pass your command with arguments as a string. The following snippet presents an example:

```
hbs::Eval {set_msg_config -suppress -id "Synth 8-6014" -string {{REPORT_PREFIX}}}
```

The `hbs::Eval` procedure has an optional `-force-in-dry` flag, which allows for forcing command evaluation in the dry run. See `'hbs doc Eval'` for more information.

The `hbs::Exec` and `hbs::ExecInCoreDir` procedures work analogously, but they execute commands instead of evaluating them. Moreover, they return the exit status, allowing you to check if commands succeed. Check `'hbs doc Exec'` and `'hbs doc ExecInCoreDir'` for more details.

An alternative approach for writing dry-run-compatible hbs files is to explicitly execute some actions in your hbs files only if the current run is not a dry run. The following snippet presents an example:

```
if {!$hbs::DryRun} {
  set_msg_config -suppress -id "Synth 8-6014" -string {{REPORT_PREFIX}}
}
```

If you use this technique, the output produced by the dry run will not create a valid script for building a project. However, this technique still allows for generating a dependency graph without running any tool flow.

Writing hbs files compatible with dry runs adds some boilerplate. Not a lot, but still. Most projects do not require hbs files compatible with dry runs. If you implement a core that you want to share with others, then it is a good idea to assume they might require dry runs. However, if you implement a project only for yourself or the company you work for, then it is advised to write dry-run-compatible hbs files only if you know you will need dry runs. If it later turns out you were wrong, and you need dry runs, you can easily adjust your hbs files. Adapting dry run incompatible hbs files is quite simple, as dry runs simply fail with an error message when they encounter an unknown command.

4.10 `test` - running testbench targets

The `test` command allows running all automatically discovered testbench targets. The `test` command is executed by the `hbs` file and requires Python to work. By default, testbench targets are run in parallel. The default number of workers equals the number of threads on your CPU. If you provide extra arguments to the `test` command, only testbench targets which path contain at least one of the provided strings are run. The following snippet presents an output for running all testbenched of the bus functional model in the VHDL APB library.

```
[user@host ap] hbs test bfm
running 4 targets with 16 workers

vhdl::amba5::apb::bfm::tb-readb   passed  warnings: 1
vhdl::amba5::apb::bfm::tb-read    passed  warnings: 1
vhdl::amba5::apb::bfm::tb-write   passed  warnings: 1
vhdl::amba5::apb::bfm::tb-writeb  passed  warnings: 1

time:     0 h 0 min 0 s
targets:  4
passed:   4
failed:   0
errors:   0
warnings: 4
```

4.11 `version` - displaying HBS version

The `version` command displays version of installed HBS. This might be helpful if the same build procedure works on one machine, but does not work on another. Based on the versoin and changelog,

you can quickly discover differences. The blow snippet shows an example output for the `version` command.

```
[user@host ~ 0] hbs version
1.0
```

4.12 `whereis` - locating cores definition

The `whereis` command allows easily locating .hbs files in which given cores are defined The `whereis` command is executed by the `hbs` file, so the command requires Python to work. The following snippet presents an example of locating core definition:

```
[user@host vsc8211-tester] hbs whereis serial-bridge
vhdl::amba5::apb::serial-bridge  /tmp/vsc8211-tester/gw/apb/apb.hbs
```

You can locate multiple cores in a single call by providing multiple arguments to the command. The following snippet presents an example:

```
[user@host vsc8211-tester] hbs whereis bridge mdio
vhdl::amba5::apb::serial-bridge  /tmp/vsc8211-tester/gw/apb/apb.hbs
vhdl::ethernet::mdio             /tmp/vsc8211-tester/gw/vhdl-ethernet/ethernet.hbs
```

5 Fetching and managing design dependencies

A lot of modern software programming languages come with official package and dependency management tools. For example, `pip` for Python, `npm` for Node.js, `cargo` for Rust. In the hardware design domain, there was never any official package and dependency manager. Keeping dependencies in-tree is de facto the standard way. In practice, people just manually or semi-automatically copy dependencies to the project sources. The dependency sources are kept in the tree of the project directory, hence the term “in-tree”. Keeping dependencies in-tree forces you to be conscious about what is included in the project. It also helps to avoid bloat.

HBS currently does not have any mechanism for fetching design dependencies. This is because different dependencies might require completely different commands to be executed to fetch them and prepare them for use. Some teams like to manage external dependencies using git submodules. Others prefer to copy dependencies manually. Yet others implement custom shell or Python scripts. HBS does not try to limit or impose anything on the user in this matter.

Although HBS does not provide any mechanism for fetching and managing design dependencies, nothing stops you from implementing such a mechanism in hbs files. You can use target procedures for that purpose if you want to track these dependencies in the target dependency graph, or you can simply define Tcl procedures outside core namespaces.

6 Tcl tips

6.1 Support for arrow keys in `tclsh`

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.

6.2 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 following example is taken directly from the hbs Tcl source code:

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

7 Examples

This section contains some examples of HBS usage. It barely presents some primary features of HBS. If you want to discover full capabilities of HBS, then run `hbs doc` command. The command lists all HBS API public symbols. To get more information on a given symbol, run `'hbs run <symbol>'` command.

7.1 Single file simulation

Let us assume we want to simulate the following minimal VHDL example:

```
entity example is end entity;

architecture test of example is
begin
  main : process is
  begin
    report "Hello from example!";
    std.env.finish;
  end process;
end architecture;
```

A minimal hbs file looks as follows:

```
namespace eval example {
  proc sim {} {
    hbs::SetTool "nvc" ;# We must set some tool.
    hbs::AddFile "example.vhd" ;# This is required, we want to simulate this file.
    hbs::SetTop "example" ;# Let the simulator know what is the top entity.
    hbs::Run ;# Run the flow for set tool.
  }
  hbs::Register ;# This is required to register the core.
}
```

Nothing from the above hbs file can be removed, except comments and preceding semicolons. The example simply won't work. Now, we can run the simulation what is shown in the following snippet:

```
[user@host test] hbs run example::sim
nvc  --std=2019 -L. --work=work -a /tmp/test/example.vhd
nvc  --std=2019 -L. -e example
nvc  --std=2019 -L. -r example --wave
** Note: writing FST waveform data to example.fst
** Note: 0ms+0: Hello from example!
   Process :example:main at /tmp/test/example.vhd:5
** Note: 0ms+0: FINISH called
   Procedure FINISH [] at ../lib/std.19/env-body.vhd:48
   Process :example:main at /tmp/test/example.vhd:8
[user@host test] echo $?
0
```

As can be seen the returned status is 0. The simulation finished successfully. The HBS always forwards the EDA tool exist status to you. Thanks to this, you can easily check if the target proc you run succeeded.

7.1.1 Placing the module in custom library

Low, let us assume we want to place our module in a custom library named `lib`. All we need is to add a call to the `hbs::SetLib` procedure. A new hbs file is presented in the following snippet:

```
namespace eval example {
  proc sim {{tool "nvc"}} {
    hbs::SetTool $tool
    hbs::SetLib "lib"; # A new call to change the library.
    hbs::AddFile "example.vhd"
    hbs::SetTop "example"
    hbs::Run
  }
  hbs::Register
}
```

The following snippet shows output from running the target:

```
[user@host test] hbs run example::sim
nvc  --std=2019 -L. --work=lib -a /tmp/test/example.vhd
nvc  --std=2019 -L. -e example
nvc  --std=2019 -L. -r example --wave
** Note: writing FST waveform data to example.fst
** Note: 0ms+0: Hello from example!
   Process :example:main at /tmp/test/example.vhd:5
** Note: 0ms+0: FINISH called
   Procedure FINISH [] at ../lib/std.19/env-body.vhd:48
   Process :example:main at /tmp/test/example.vhd:8
```

As can be seen, the library has been changed to `lib` (`--work=lib`).

7.1.2 Changing the simulator via command line

Low, let us assume we want to be able to easily change the simulator. We would like to define the simulator when executing the target. The simplest way is to add a parameter to the target. The following snippet presents modified hbs files:

```
namespace eval example {
  proc sim {{tool "nvc"}} {
    hbs::SetTool $tool ;# Now we use proc parameter to set desired tool.
    hbs::AddFile "example.vhd"
    hbs::SetTop "example"
    hbs::Run
  }
  hbs::Register
}
```

The parameter is named `tool` and has the default value equal `"nvc"`. The following snippet presents how to run simulation using different simulators:

```
[user@host test] hbs run example::sim
nvc  --std=2019 -L. --work=work -a /tmp/test/example.vhd
nvc  --std=2019 -L. -e example
nvc  --std=2019 -L. -r example --wave
** Note: writing FST waveform data to example.fst
** Note: 0ms+0: Hello from example!
   Process :example:main at /tmp/test/example.vhd:5
** Note: 0ms+0: FINISH called
   Procedure FINISH [] at ../lib/std.19/env-body.vhd:48
   Process :example:main at /tmp/test/example.vhd:8

[user@host test] hbs run example::sim ghdl
ghdl -a  --std=08  -Pwork --work=work --workdir=work  /tmp/test/example.vhd
ghdl -e  --std=08 --workdir=work  -Pwork  example
./example  --wave=example.ghw
/tmp/test/example.vhd:7:5:@0ms:(report note): Hello from example!
simulation finished @0ms

[user@host test] hbs run example::sim xsim
xvhdl  -work work --2008 /tmp/test/example.vhd
INFO: [VRFC 10-163] Analyzing VHDL file "/tmp/test/example.vhd" into library work
INFO: [VRFC 10-3107] analyzing entity 'example'
...
## exit
INFO: xsimkernel Simulation Memory Usage: 277636 KB (Peak: 335168 KB), Simulation
CPU Usage: 630 ms
INFO: [Common 17-206] Exiting xsim at Tue Dec 23 11:44:03 2025...
```

The log from the `xsim` is very verbose, that is why it has been trimmed.

7.2 Tool and tool type specific actions

Let us assume we have a 2-stage flip-flop clock domain crossing synchronizer. The synchronizer module consists of two files:

1. `synchronizer.vhd` - synchronizer hardware description,
2. `vivado-constr.xdc` - Vivado constraints for the synchronizer.

The constraint file must be scoped to the synchronizer module. However, Vivado does not allow scoping the `.xdc` file from within this file. The `SCOPED_TO_REF` property must be set by explicitly calling Vivado `set_property` command from the Tcl shell level. This is extremely simple in HBS, as hbs files are executed directly by the EDA tool's embedded Tcl shell. The following listing presents the synchronizer core definition in hbs file:

```
namespace eval cdc::synchronizer {
  proc src {} {
    hbs::AddFile synchronizer.vhd
    # Check if current EDA tool is Vivado.
    if {[string match "vivado*" $hbs::Tool]} {
      hbs::AddFile vivado-constr.xdc
      set_property SCOPED_TO_REF Synchronizer [get_files vivado-constr.xdc]
    } elseif {$hbs::ToolType eq "synthesis"} {
      error "Synchronizer entity misses constraint file for $hbs::Tool"
    }
  }
  hbs::Register
}
```

The constraint file is added to the project and scoped to the module only if the set EDA tool is Viavdo (if branch). Moreover, if you want your core to be potentially used on platforms from other vendors, but you do not know how to write proper constraints for other tools, you can issue an error (elseif branch). If someone else tries to utilize your core with a different EDA tool, they will get a meaningful error message even before their project starts building. Please note that the error is issued only if the set EDA tool is used for synthesis. The constraint file is not required for simulation.

HBS tracks the type of set EDA tool in the `hbs::ToolType` variable. This is very useful, as some cores might require one set of build actions for simulation, and completely different set of actions for synthesis. To obtain more information on tool types run `'hbs doc ToolType'` in the shell.

7.3 Sharing actions between core targets

It is very common that multiple targets of the same core share most of their actions. For example, core testbench targets use mostly the same files and set the same simulator options. To share common actions between core targets, you can define an extra procedure within the core namespace. As this additional procedure is not a target per se, you can start its name with the underscore character (_). Thanks to this, HBS will not treat the extra procedure as a core target procedure. For testbench targets, it is common to simply name the helper procedure "`_tb`". The following snippet presents an example.

```
namespace eval vhdl::tinyuart {
  proc src {} {
    hbs::SetLib "tinyuart"
    hbs::AddFile tinyuart.vhd
  }
  proc _tb {top} {
    hbs::SetTool "nvc"
    hbs::AddPostElabCb hbs::SetArgsPrefix "--messages=compact"
    hbs::SetTop $top
    src
    hbs::SetLib "" ;# Place testbench entity in the default library.
    hbs::AddFile [string map {_ -} $top].vhd ;# Replace '_' with '-' in file name.
  }
  # Transmitter testbench
  proc tb-tx {} {
    _tb "tb_tx"
    hbs::Run
  }
  # Receiver testbench
  proc tb-rx {} {
```

```
    _tb "tb_rx"
    hbs::Run
  }
  # Loopback testbench
  proc tb-loopback {} {
    _tb "tb_loopback"
    hbs::Run
  }
  hbs::Register
}
```

The `vhdl::tinyuart` core has three testbench targets. They all utilize the same simulator with the same command line options. Moreover, all testbenches require core source files for simulation, hence the call to the `src` procedure in the `_tb` procedure. Testbench targets only pass the testbench top entity name to the `_tb` procedure and call `hbs::Run`.

7.4 More examples

If you have some interesting exmaple of HBS usage, feel free to prepare a pull request with extension the following list:

- VHDL APB library,
- VSC8211 Tester.

8 Contributing

You can contribute to HBS directly on https://github.com/m-kru/hbs.