FuseSoC Documentation

Release 1.12.0

Olof Kindgren

Feb 25, 2021

CHAPTERS

FuseSoC is a build system for digital hardware (e.g. Verilog or VHDL designs), and a package manager for reusable blocks in hardware designs.

This documentation contains material for different audiences.

The *User Guide* explains how to get started with FuseSoC, starting from the installation.

The *Reference Guide* provides a detailled description of all file formats and APIs.

The *Developer's Guide* is aimed at developers of FuseSoC itself. It explains how to set up a development environment, how the source code is structured, and how patches and bug reports can be submitted to the project.

CHAPTER

ONE

FUSESOC USER GUIDE

The FuseSoC User Guide is aimed at hardware developers utilizing FuseSoC to build and integrate their hardware designs.

Learn how to use FuseSoC in an existing project.

Have you checked out a hardware design project that uses FuseSoC and are trying to understand how to build the design? Get started by *installing FuseSoC*, and then have a look at the *usage documentation*.

Add FuseSoC support to your hardware project.

If you are starting a new hardware design project, or already have source files and are looking for a better way to build your project and integrate third-party components? Get started by *installing FuseSoC*, read a bit about the *concepts and terminology of FuseSoC*, and then move on to *add FuseSoC core description files* to your project.

1.1 Why FuseSoC?

FuseSoC is an award-winning package manager and a set of build tools for HDL (Hardware Description Language) code.

Its main purpose is to increase reuse of IP (Intellectual Property) cores and be an aid for creating, building and simulating SoC solutions.

FuseSoC makes it easier to

- reuse existing cores
- create compile-time or run-time configurations
- run regression tests against multiple simulators
- Port designs to new targets
- let other projects use your code
- set up continuous integration

FuseSoC is non-intrusive Most existing designs doesn't need any changes to work with FuseSoC. Any FuseSoC-specific patches can be applied on the fly during implementation or simulation

FuseSoC is modular It can be used as an end-to-end flow, to create initial project files for an EDA tool or integrate with your custom workflow

FuseSoC is extendable Latest release support simulating with GHDL, Icarus Verilog, Isim, ModelSim, Verilator and Xsim. It also supports building FPGA images with Altera Quartus, project IceStorm, Xilinx ISE and Xilinx Vivado. Support for a new EDA tool requires ~100 new lines of code and new tools are added continuously

FuseSoC is standard-compliant Much effort has gone into leveraging existing standards such as IP-XACT and vendor-specific core formats where applicable.

FuseSoC is resourceful The standard core library currently consisting of over 100 cores including CPUs, peripheral controllers, interconnects, complete SoCs and utility libraries. Other core libraries exist as well and can be added to complement the standard library

FuseSoC is free software It puts however no restrictions on the cores and can be used to manage your company's internal proprietary core collections as well as public open source projects

FuseSoC is battle-proven It has been used to successfully build or simulate projects such as Nyuzi, Pulpino, VScale, various OpenRISC SoCs, picorv32, osvvm and more.

1.2 Installing FuseSoC

FuseSoC is written in Python and runs on all major operating systems.

1.2.1 System Requirements

Before installing FuseSoC check your system requirements.

- Operating System: Linux, Windows, macOS
- Python 3.5 or newer. (The last version supporting Python 2.7 is FuseSoC 1.10.)
- The Python packages `setuptools` and `pip` need to be installed for Python 3.

1.2.2 Installation under Linux

Note: Do not type the `$` symbol shown in the instructions below. The symbol indicates that the command is to be typed into a terminal window. Lines not prefixed with `$` show the output of the command. Depending on your system, the output might be different.

FuseSoC is provided as `fusesoc` Python package and installed through pip, the Python package manager. The steps below cover the most common installation cases. Refer to the pip documentation for more advanced installation scenarios.

FuseSoC, like all Python packages, can be installed for the current user, or system-wide for all users. The system-wide installation typically requires root permissions.

Installation for the current user

To install the current stable version of FuseSoC for the current user, open a terminal window and run the following command. If an older version of FuseSoC is found, this version is upgraded to the latest stable release.

```
$ pip3 install --upgrade --user fusesoc
```

Check that the installation worked by running

```
$ fusesoc --version
1.12.0
```

If this command works FuseSoC is installed properly and ready to be used.

If the terminal reports an error about the command not being found check that the directory `~/.local/bin` is in your command search path (`PATH`), or perform a system-wide installation instead (see below).

System-wide installation

FuseSoC can be installed for all users on a system. This operation typically requires root permissions.

```
$ sudo pip3 install --upgrade fusesoc
```

Uninstalling FuseSoC

Use `pip` to remove FuseSoC from your system.

```
$ pip3 uninstall fusesoc
```

1.2.3 Installation under Windows

Todo: Add Windows installation instructions.

1.2.4 Installation under macOS

Todo: Add macOS installation instructions.

1.3 Understanding FuseSoC

1.3.1 The components of FuseSoC

FuseSoC is a package manager and a build system for HDL code. The following sections explain the main concepts of FuseSoC, and how they work together to obtain, configure, and build hardware designs.

A fundamental entity in FuseSoC is a core. Cores can be discovered by the FuseSoC package manager in local or remote locations, and combined into a full hardware design by the build system.

FuseSoC's basic building block: cores

A **FuseSoC core** is a reasonably self-contained, reusable piece of IP, such as a FIFO implementation. In other contexts, the same concept is called package (as in deb package, npm package, etc.), or programming library (a term which is used with a different meaning in FuseSoC).

A core in FuseSoC has a name, such as `example:ip:fifo`. The name and other information about the core, such as the list of (e.g. Verilog or VHDL) source files, is contained in a description file called the **core file**. Core files have filenames ending in `.core`.

A core can also have dependencies on other cores. For example, a "FIFO core" can have a dependency on an "SRAM core."

Discover cores: the package manager

FuseSoC cores can be stored in many places, both locally and remote. Finding cores, and producing a list of cores available to the user, is the job of the FuseSoC package manager.

To find cores, the FuseSoC package manager searches through a configurable set of **core libraries**. A core library can be local or remote. In the simplest case, a local core library is just a directory with cores, as it is common in many hardware projects. To support more advanced cases of code re-use and discovery, FuseSoC can also use remote core libraries. Remote core libraries can be a key enabler to IP re-use especially in larger corporate settings, or in free and open source silicon projects.

From cores to a whole system: the build system

The FuseSoC build system resolves all dependencies between cores, starting from a top-level core. A top-level core is technically just another FuseSoC core, but with a special meaning: it is the entry point into the design. The output of the dependency resolution step is a list of source files and other metadata that an EDA tool needs to build (i.e. synthesize, simulate, lint, etc.) the top-level design. The dependency resolution process can be influenced by constraints. Constraints are effectively input variables to the build process, and capture things like the target of the build (e.g. simulation, FPGA synthesis, or ASIC synthesis), the EDA tool to be used (e.g. Verilator or Xilinx Vivado), and much more.

From a file list to a synthesized design: EDAlize it!

After the build system has collected all source files and parameters of the design, it is time to hand off to the build tool, such as Xilinx Vivado for FPGA synthesis, Verilator for Verilog simulation, and so on.

How exactly the hand-off is performed is highly tool-dependent, but FuseSoC abstracts these differences away and users typically don't need to worry about them. For example, in for Vivado, FuseSoC creates a Vivado project file and then executes Vivado to run through the synthesis and place and route steps until a final bitstream is produced. For Verilator, it creates a Makefile and then calls Verilator to do its job.

FuseSoC supports many of the proprietary and open source EDA tools commonly used, and can be easily extended to support further ones.

1.3.2 Concepts of the FuseSoC build system

To understand how FuseSoC builds a design it is necessary to understand three basic concepts: targets, tool flows, and build stages.

Tool flows

A tool flow (often abbreviated to just "tool" or "EDA tool") is a piece of software operating on the design to analyze it, simulate it, transform it, etc. Common categories of tools are simulators, synthesis tools, or static analysis tools. For example, Verilator is a tool (a simulation and static analysis tool), as is Xilinx Vivado (an FPGA tool flow), or Synopsys Design Compiler (an ASIC synthesis tool).

FuseSoC tries its best to hide differences between tools to make switching between them as easy as possible. For example, it often only requires a different command line invocation of FuseSoC to simulate a design with Synopsys VCS instead of with Icarus Verilog. Of course, customization options for individual tools are still available for when they are needed.

Targets

Many things can be done with a hardware design: it can be synthesized for an FPGA, it can be simulated, it can be analyzed by lint tools, and much more. Even though all of these things operate on the same hardware design, there are differences: design parameters (e.g. Verilog defines and parameters, or VHDL generics) are set differently, source files are added or substituted (e.g. a top-level testbench wrapper is added, or a behavioral model of an IP block is exchanged against a hard macro), etc. In FuseSoC, a target is a group of such settings. Users of FuseSoC can freely name targets. Commonly used targets are one for simulation (typically called `sim`), one for FPGA or ASIC synthesis (`synth`), and one for static analysis (`lint`).

Build stages

FuseSoC builds a design in three stages: setup, build, and run.

1. The **setup** stage. In this first step, the design is stitched together and prepared to be handed over to the tool flow.
    1. A dependency tree is produced, starting from the top-level core.
    2. Generators (special cores with dynamic behavior) are called. Cores produced by generators are inserted into the dependency tree as well.
    3. The dependency tree is resolved to produce a flattened view of the design. All design information is written into an EDAM file.
    4. Tool-flow specific setup routines are being called.
2. The **build** stage runs the tool flow until some form of output file has been produced.
3. The **run** stage somehow "executes" the build output. What this exactly means is highly tool flow dependent: for simulation flows, the simulation is executed. For static analysis (lint) flows, the lint tool is called and its output is displayed. For FPGA flows, the FPGA is programmed with the generated bitstream.

1.4 Running FuseSoC

FuseSoC is a command-line tool; this section explains how to use it. The following content is aiming at users who already have a hardware design which uses FuseSoC.

1.4.1 Build a design

The `fusesoc run` group of commands is used to setup, build, and (if possible) run a design. The exact actions taken by the individual steps depend on the toolflow.

```
usage: fusesoc run [-h] [--no-export] [--build-root BUILD_ROOT] [--setup] [--build] [-
↪-run] [--target TARGET] [--tool TOOL] [--flag FLAG] [--system-name SYSTEM_NAME]␣
↪system ...

positional arguments:
  system                Select a system to operate on
  backendargs           arguments to be sent to backend

optional arguments:
  -h, --help            show this help message and exit
  --no-export           Reference source files from their current location instead of␣
↪exporting to a build tree
  --build-root BUILD_ROOT
```

(continues on next page)

(continued from previous page)

```
                        Output directory for build. Defaults to build/$VLNV
  --setup               Execute setup stage
  --build               Execute build stage
  --run                 Execute run stage
  --target TARGET       Override default target
  --tool TOOL           Override default tool for target
  --flag FLAG           Set custom use flags. Can be specified multiple times
  --system-name SYSTEM_NAME
                        Override default VLNV name for system
```

1.5 Building a design with FuseSoC

The FuseSoC build system pieces together a hardware design from individual cores.

Building a design in FuseSoC means *calling a tool flow to produce some output, and execute it.* Depending on the *target* and the *tool flow* chosen, the build process can do and produce very different things: it could produce a runnable simulation, an FPGA bitstream, or run a static analysis tool to check for common programming errors.

To build a hardware design with FuseSoC two steps are required:

1. Write one or multiple FuseSoC core description file(s). Read on in this section for information on how to do that.
2. Call `fusesoc run`. FuseSoC is a command-line tool and accessible through the `fusesoc` command. See *Running FuseSoC* for information on how to use the `fusesoc` command.

Typically, FuseSoC support can be added to an existing design without changes to the directory structure or the source files. The following sections explain how to do it.

The first three sections are recommended reading for all users of FuseSoC. The first section *Writing core files* is an introduction into *core description files* and how to write them. The second and third section, *Passing options to tools* and *Dependencies: link cores together for re-use* look at how to customize what the (EDA) *tools* are doing, and how cores can be combined to form a larger system.

The subsequent sections are advanced topics, which are only relevant in some projects.

A full reference documentation on the CAPI2 core file format can be found in the section *CAPI2 Reference*.

1.5.1 Writing core files

A *core* is described in a core description file, or core file.

Core files are written in *YAML* syntax and follow the FuseSoC's own CAPI (version 2) schema, which describes the structure of core files (e.g. which keys and values are allowed where). Don't worry: using FuseSoC neither requires a full understanding of YAML, nor an up-front knowledge of CAPI. However, some key facts about YAML are important.

Things one should know about YAML

- **Whitespace matters** (as in Python): indentation is used to group settings together to form a hierarchy. The exact amount of whitespace used for indentation does not matter; typically two or four spaces are used.
- Think of a YAML file as a **hierarchical, typed data structure**. There are lists, dictionaries (key/value pairs), integers, strings, etc.
- YAML syntax provides **multiple ways to describe the same structure**. It does not matter to FuseSoC which syntax variant is used. For example, a list of items can be written in the following two, semantically identical ways.

```
[ item1, item2 ]
```

  is semantically identical to

```
- item1
- item2
```

  The same is true for dictionaries (key/value pairs).

```
{ key1: value1, key2: value2 }
```

  is semantically identical to

```
key1: value1
key2: value2
```

  In most cases, the longer (second) form is preferred, as it is easier to make changes while keeping the diff easy to read.

For a quick introduction into most of YAML's features have a look at Learn YAML in Y minutes. The full YAML 1.2 specification is available at yaml.org (it's not an easy read, though).

An example: the blinky core

The following sections explain how to add FuseSoC support to a hardware project. The code is taken from an example design in the FuseSoC source tree in the `tests/userguide/blinky` directory.

The design consists of two SystemVerilog files, a testbench, a Xilinx constraint file (with pin mappings for a Nexys Video FPGA board), and finally, the FuseSoC core file.

```
$ tree tests/userguide/blinky/
tests/userguide/blinky/
├── blinky.core
├── data
│   └── nexys_video.xdc
├── rtl
│   ├── blinky.sv
│   └── macros.svh
└── tb
    └── blinky_tb.sv

3 directories, 5 files
```

To get started, here's the full `blinky.core` file. The following sections will refer back to this example to discuss it in detail.

Listing 1: `blinky.core`, an exemplary core file

```
CAPI=2:
name: fusesoc:examples:blinky:1.0.0
description: Blinky, a FuseSoC example core

filesets:
  rtl:
    files:
      - rtl/blinky.sv
      - rtl/macros.svh:
          is_include_file: true
    file_type: systemVerilogSource

  tb:
    files:
      - tb/blinky_tb.sv
    file_type: systemVerilogSource

  nexys_video:
    files:
      # YAML short form, see rtl/macros.svh above for the longer form.
      - data/nexys_video.xdc: {file_type: xdc}

targets:
  # The "default" target is special in FuseSoC and used in dependencies.
  # The "&default" is a YAML anchor referenced later.
  default: &default
    filesets:
      - rtl
    toplevel: blinky
    parameters:
      - clk_freq_hz

  # The "sim" target simulates the design. (It could have any name.)
  sim:
    # Copy all key/value pairs from the "default" target.
    <<: *default
    description: Simulate the design
    default_tool: icarus
    filesets_append:
      - tb
    toplevel: blinky_tb
    tools:
      icarus:
        iverilog_options:
          - -g2012 # Use SystemVerilog-2012
      modelsim:
        vlog_options:
          - -timescale=1ns/1ns
    parameters:
      - pulses=10

  # The "synth" target synthesizes the design. (It could have any name.)
  synth:
    <<: *default
    description: Synthesize the design for a Nexys Video FPGA board
```

(continues on next page)

(continued from previous page)

```
    default_tool: vivado
    filesets_append:
      - nexys_video
    tools:
      vivado:
        part: xc7a200tsbg484-1
    parameters:
      - clk_freq_hz=100000000

parameters:
  clk_freq_hz:
    datatype    : int
    description : Frequency of the board clock, in Hz
    paramtype   : vlogparam
  pulses:
    datatype    : int
    description : Number of pulses to run in testbench
    paramtype   : vlogparam
```

Naming the core file

The core file can have any name, but it must end in `.core`. It is recommended to choose a file name matching the core name, as discussed below.

The first line: `CAPI=2`

A core file always starts with the line `CAPI=2`. No other content (including comments) is allowed before this line, as FuseSoC uses this line to differentiate between different versions of the CAPI schema. Only CAPI version 2 is specified at the moment.

The core name, version, and description

Each core has a name, given in the `name` key. Core names can be freely chosen, but need to follow a common structure called *VLNV*. *VLNV* stands the four parts of a core name, which are separated by colon (`:`): Vendor, Library, Name, and Version.

Version numbers should be three numbers in the form major.minor.patch and follow *semantic versioning* (SemVer).

Cores can also have a description, given in the `description` key. A description is optional, but recommended.

```
name: fusesoc:examples:blinky:1.0.0
description: Blinky, a FuseSoC example core
```

In this example, the vendor is `fusesoc`, the library is `examples`, and the name of the core is `blinky`. The version is set to `1.0.0`.

Specifying source files

A core typically consists of one or multiple source files. Source files are grouped into file sets under the `filesets` key.

FuseSoC does neither mandate a specific grouping, nor naming of file sets. It is common to use one file set for RTL (design) files, and one for testbench files.

The following example shows a single file set, `rtl`, with a set of common keys.

```
filesets:
  rtl:
    files:
      - rtl/blinky.sv
      - rtl/macros.svh:
          is_include_file: true
    file_type: systemVerilogSource
```

For each named file set, several keys are supported:

- `files`: An ordered list of source files. The list of source files is ordered: the files will be passed to the tool in exactly the given order. This is important, for example, in SystemVerilog, where packages need to be compiled before they can be used by subsequent source files.
- `file_type`: The default file type for all files in the `files` list.
- `depend`: Dependencies on other cores. Dependencies are explained in depth at *Dependencies: link cores together for re-use*.

Source files

Source files are resolved relative to the location of the core file and must be stored in the same directory as the core file, or in a subdirectory of it. Source file names cannot be absolute paths, or start with `../`.

Optionally, source files can have attributes; the file `macros.vh` is an example of that. When specifying attributes, end the file name with a colon (`:`), and specify attributes as key-value pairs below it. (Alternatively, the equivalent short form syntax can be used, e.g. `macros.vh: {is_include_file: true}`.)

The most common attributes are:

- `is_include_file`: The file is an include file. In Verilog and C/C++, this means the file is not passed to the tool directly, but instead the file is included by another source file. FuseSoC ensures that the tool finds the include file, e.g. by passing an appropriate include path to the tool.
- `file_type`: Override the default file type of the fileset for this particular file.

Refer to the *CAPI2 reference documentation* for more details.

File types

A file type describes the type of source file. FuseSoC does not use this information itself, but passes it on to tool backends which then configure the tool appropriately depending on the file type encountered.

Commonly used file types are:

- `verilogSource`: Verilog source code, up to Verilog-2001. Files ending in `.v` or `.vh` should use this type.
- `systemVerilogSource`: SystemVerilog source code (design and test code). Files ending in `.sv` or `.svh` should use this type.

- `vhdlSource`: VHDL source code. Files ending in `.vhd` or `.vhdl` should use this file type.

Refer to the *CAPI2 reference documentation* for more details.

Targets

A *target* can be seen as something you would like to do with the source code in the core: synthesize it, simulate it, lint it. Targets are specified as dictionaries under the `targets` top-level key.

```
targets:
  # The "default" target is special in FuseSoC and used in dependencies.
  # The "&default" is a YAML anchor referenced later.
  default: &default
    filesets:
      - rtl
    toplevel: blinky
    parameters:
      - clk_freq_hz

  # The "sim" target simulates the design. (It could have any name.)
  sim:
    # Copy all key/value pairs from the "default" target.
    <<: *default
    description: Simulate the design
    default_tool: icarus
    filesets_append:
      - tb
    toplevel: blinky_tb
    tools:
      icarus:
        iverilog_options:
          - -g2012 # Use SystemVerilog-2012
      modelsim:
        vlog_options:
          - -timescale=1ns/1ns
    parameters:
      - pulses=10

  # The "synth" target synthesizes the design. (It could have any name.)
  synth:
    <<: *default
    description: Synthesize the design for a Nexys Video FPGA board
    default_tool: vivado
    filesets_append:
      - nexys_video
    tools:
      vivado:
        part: xc7a200tsbg484-1
    parameters:
      - clk_freq_hz=100000000
```

The blinky example shown above defines three targets: the `default` target, a `sim` target to simulate the design, and a `synth` target to synthesize it. Many designs also define a `lint` target to run static analysis jobs. The `sim` and `synth` targets are optional and could have had any name. The `default` target is special and required.

Within a target

Within each target block multiple keys determine what the target does. The most common keys are:

- `filesets`: An ordered list of file sets (source files) included in the target.
- `description` (optional): A description of the target.
- `toplevel` (optional): The name of the design toplevel. (For advanced scenarios it is possible to specify a list of multiple toplevels instead of just a single one.)
- `default_tool` (optional): The default tool to be used to build the target. The tool can also be set or overridden through a FuseSoC command-line argument.
- `tools` (optional): Tool-specific settings, grouped by tool name.
- `parameters` (optional): Parameters (Verilog parameters and defines, VHDL generics, etc.) to be passed to the design, or forced to a certain value.

The `filesets_append` key is part of an inheritance schema and explained further in section *Inheritance and the default target*.

The `default` target

The `default` target is the only required target. It serves two purposes:

- The `default` target is used if no other target is explicitly selected when running FuseSoC.
- The contents of the `default` target are used if the core is used as dependency (described in detail in *Dependencies: link cores together for re-use*).

All reusable code in the core should go into the `default` target: RTL files, lint waivers, reusable constraints, etc.

Inheritance and the `default` target

Importantly, other targets in the same core do *not* inherit the contents of the `default` target automatically. To achieve such inheritance behavior, FuseSoC provides a flexible inheritance mechanism, based on YAML anchors/references, YAML `<<` merge operator, and a FuseSoC-specific list append feature.

The *blinky.core* shows the recommended template to inherit configuration between targets.

1. Add `&default` after the `default:` text. This defines a YAML anchor named `default`, which can be referenced later in the file.
2. Add a line `<<: *default` to the target where you want to inherit from `default` (the `sim` and `synth` targets in the example code). This line will effectively "copy over" all configuration under the `default` target.

As always with inheritance the interesting questions are around overriding behavior.

- Settings (keys) given in the target which inherits from `default` override the keys in `default`. For example, the `toplevel` key in the `sim` target is overridden to be `tb`. Note that no merging of setting data structure is performed.
- For settings which are lists, for example the `filesets` key, FuseSoC provides a way to combine lists by adding `_append` to the name of the key.

  This behavior is best explained by example. The `filesets` list in the `default` target consists of a single item, `rtl`. The `sim` target wants to append the item `tb` (a file set with testbench files) to the list. To do so, it specifies the special `filesets_append` key with a partial list. When evaluating the core file, FuseSoC

appends the contents of `sim.fileset_append` at the end of `default.fileset` to form a list with two items: `rtl`, and `tb`. The same behavior works for all lists in core files.

1.5.2 Passing options to tools

Note: Refer to the *CAPI2 reference documentation* for a list of all available tools and their options.

FuseSoC abstracts away many differences between *tools* and tries to provide sane defaults to build many designs out of the box with no further configuration required. However, not all tool-specific details can be hidden. At the same time, a certain level of tool-specific configurability is required to make full use of the features available in different tools. Tool options are FuseSoC's way of customizing the way tools are used to build the design.

When calling `fusesoc run` on the command line any tool can be chosen to build a design with the `--tool` argument. If no tool-specific configuration is given in the core file, the default tool configuration is used, which might or might not work for a given design.

To customize tool behavior a tool-specific section can be added to a core file at `targets.TARGETNAME.tools.TOOLNAME`. The name of the tool (`TOOLNAME`) must match FuseSoC's internal tool name (as passed to `fusesoc run --tool=TOOLNAME`). Depending on the tool different options are available. Refer to the *CAPI2 reference documentation* for a list of all available tools and their options.

Most tool backends provide a way to set command line options to influence how the tools are called. Typically, these keys are called `BINARYNAME_options`, and they take a list of arguments as value.

The example below shows how tool options for Icarus Verilog (`icarus`) and Mentor ModelSim (`modelsim`) are set.

- The `iverilog` binary will be called with the `-g2012` command-line argument, indicating that SystemVerilog 2012 support should be enabled.
- Similarily, for ModelSim the argument `-timescale=1ns/1ns` will be passed to the `vlog` binary, which elaborates the design.

```
# A fragment from blinky.core
# ...
targets:
  sim:
    # ...
    tools:
      icarus:
        iverilog_options:
          - -g2012 # Use SystemVerilog-2012
      modelsim:
        vlog_options:
          - -timescale=1ns/1ns
```

Note: Where to find tool-specific code in FuseSoC

The tool-specific code is provided by the edalize library. Most files, such as project files and Makefiles, are templates within edalize and can be improved easily if necessary. Please open an issue at the edalize issue tracker on GitHub to suggest improvements to tool-specific code.

1.5.3 Dependencies: link cores together for re-use

For a long time, productivity gains in hardware designs have been achieved primarily by re-using existing code, a.k.a. IP blocks. Re-use is also engrained into FuseSoC: re-usable hardware design components are packaged into cores and then used in other designs. In FuseSoC (and many other package managers), re-use is achieved by expressing dependencies between FuseSoC cores.

This section explains how dependencies are specified, how they are resolved by FuseSoC, and how they can be constrained.

A dependency example: DualBlinky

We introduced the basic FuseSoC features by creating an reusable core called Blinky. To illustrate the concept of dependencies in FuseSoC we employ another example: DualBlinky, the "dual-core" version of Blinky. Again, all source code is available in the FuseSoC source tree in the `tests/userguide/dualblinky` directory.

```
$ tree tests/userguide/dualblinky
tests/userguide/dualblinky
├── data
│   └── nexys_video.xdc
├── dualblinky.core
└── rtl
    └── dualblinky.sv

2 directories, 3 files
```

The core file is shown in full below.

Listing 2: `dualblinky.core`, 2x Blinky

```
CAPI=2:
name: fusesoc:examples:dualblinky:1.0.0
description: DualBlinky, a FuseSoC example with dependencies

filesets:
  rtl:
    files:
      - rtl/dualblinky.sv
    file_type: systemVerilogSource
    depend:
      - fusesoc:examples:blinky

  nexys_video:
    files:
      - data/nexys_video.xdc:
          file_type: xdc

targets:
  default: &default
    filesets:
      - rtl
    toplevel: dualblinky

  synth:
    <<: *default
    description: Synthesize the design for a Nexys Video FPGA board
```

(continues on next page)

(continued from previous page)

```
    default_tool: vivado
    filesets_append:
      - nexys_video
    tools:
      vivado:
        part: xc7a200tsbg484-1
    parameters:
      clk_freq_hz: 100000000

parameters:
  clk_freq_hz:
    datatype: int
    description: Frequency of the board clock, in Hz
    paramtype: vlogparam
```

Specifying a dependency

Dependencies in FuseSoC are expressed between a file set and a core. They are listed in a *core file* as in the `filesets.FILESET_NAME.depend` section.

The example below shows how to create a dependency between the `fusesoc:examples:blinky` core and the `rtl` file set of the `fusesoc:examples:dualblinky:1.0` core.

```
# Excerpt of dualblinky.core

# ...

filesets:
  # ...
  rtl:
    # ...
    depend:
      - ">=fusesoc:examples:blinky:1.0"
```

Note: YAML requires quotation marks for strings with special characters, as they are used in version constraints. Both single (`'`) and double quotes (`"`) can be used.

File ordering

File ordering (compilation order) is important in many hardware design projects. The following rules apply.

- Files from dependencies are inserted into the file list before the files in the file set where the dependency is declared.
- The order in which dependencies are listed in the `depend` section does not imply any ordering. That is, specifiying `depend: [A, B]` does not guarantee that files from core `A` are included before the ones from core `B`. (If such an order is desired, make core `B` depend on `A`.)

What happens if a dependency is specified?

Declaring a dependency includes the dependent core in the build. More specificially, the following sections specified in the `default` target of the dependent core are included:

- `filesets`: File sets to include.
- `hooks`: A list of hooks to execute.
- `generate`: List of generators.
- `parameters`: List of available parameters.
- `vpi`: List of VPI objects.

Notably *not* included are the `tools`, `toplevel`, `description`, and `default_tool` sections of the `default` target. Also, no target other than `default` is considered when including a dependency.

Version constraints

Version constraints specify which version of a dependent core can be used, and which versions are incompatible.

Within a *core file*, version constraints are expressed by prefixing a core name with a version comparision operator. The following version comparison operators are available.

Table 1: Version comparison operators

Operator	Meaning	Example
=	exactly	`=fusesoc:examples:blinky:1.2`: exactly version 1.2
<	less (lower) than	`<fusesoc:examples:blinky:1.2`: any version before 1.2, e.g. 0.9
<=	at most (less than or equal to)	`<=fusesoc:examples:blinky:1.2`: at most version 1.2, e.g. 0.9, or 1.2
>=	at least (greater than or equal to)	`>=fusesoc:examples:blinky:1.2`: version 1.2, or any newer version, e.g. 1.2, or 10.7
>	more (higher) than	`>fusesoc:examples:blinky:1.2`: any version after 1.2, e.g. 1.3, or 10.7
^	Caret requirement: any version less than the next major version (see below)	`^fusesoc:examples:blinky:1.2`: >=1.2.0 <1.3.0
~	Tilde requirement: allow updates to the current version (see below)	`~fusesoc:examples:blinky:1.2`: >=1.2.0 <2.0.0

Notes:

- If no operator is specified, then = is assumed. So the = operator is effectively optional.
- If no version number is given any version is accepted, i.e. `>= 0.0.0`.

Caret requirements

Caret requirements allow semantic versioning-compatible updates to a specified version. An update is allowed if the new version number does not modify the left-most non-zero digit in the major, minor, patch grouping.

Tilde requirements

Tilde requirements specify a minimal version with some ability to update. If you specify a major, minor, and patch version or only a major and minor version, only patch-level changes are allowed. If you only specify a major version, then minor- and patch-level changes are allowed.

Semantic versioning (SemVer)

A common scenario when declaring dependency is the following: "Core B depends on a version of core A which has the same interface as version 1.0.0, but may contain additional bug fixes in the implementation of that interface." Version numbers and dependencies alone cannot express this relationship, as they are (by default) meaningless. Equally, tools have a very hard time determining such compatibility accurately. Instead, humans are needed to attach meaning to version numbers, and that's where semantic versioning comes in.

Semantic versioning is a convention that gives meaning to version numbers. Being a convention, semantic versioning is not enforced by tooling, but relies on cooperation and a shared understanding between authors of reusable IP cores. Effectively, semantic versioning allows authors to encode in the version number information such as "this version breaks API compatibility", "this version is backwards compatible with a certain previous version", etc.

A detailed explanation of semantic versioning is available at semver.org. The basics, however, are quickly explained. Semantic versioning expects version numbers with three components, MAJOR, MINOR, and PATCH, such as 1.0.3. With this structure in place, follow these guidelines:

> Given a version number MAJOR.MINOR.PATCH, increment the:
>
> 1. MAJOR version when you make incompatible API changes,
> 2. MINOR version when you add functionality in a backwards compatible manner, and
> 3. PATCH version when you make backwards compatible bug fixes.
>
> Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.
>
> —Semantic Versioning 2.0.0 (Summary)

1.5.4 Flags: constraints in dependencies

Todo: Document flags.

1.5.5 Generators: produce and specialize cores on demand

Todo: This section was taken from older documentation and needs to be adjusted in style and content for the refactored user guide.

FuseSoC core files lists files which are natively used by the backends, such as VHDL/(System)Verilog files, constraints, tcl scripts, hex files for $readmemh, etc. There are however many cases where these files need to be created from another format. Examples of this are Chisel/MyHDL/Migen source code which output verilog, C programs that are compiled into a format suitable to be preloaded into memories or any kind of description formats used to create HDL files. For these cases FuseSoC supports generators, which is a mechanism to generate core files on the fly during the FuseSoC build flow. Since there are too many custom programs to generate HDL files, it is not feasible to have them all inside of FuseSoC. Instead they are implemented as stand-alone programs residing within cores, which can be invoked by FuseSoC. The generators support consists of three parts:

- The generator itself, which is a stand-alone program residing inside a core. It needs to accept a yaml file with a defined structure described below as its first (and only) argument. The generator will output a valid .core file and output files in the directory where it is called.
- The core that contains a generator must define a section in its core file to let FuseSoC know that it has a generator and how to invoke it.
- A core using a generator must contain a section that describes which parameters to send to the generator, and each target must list which generators to use

Creating a generator

Generators can be written in any language. The only requirement is that they are executable on the command line and accepts a yaml file for configuration as its first argument. This also means that generators can be used outside of FuseSoC to create cores. The yaml file contains the configuration needed for the generator. The following options are defined for the configuration file.

Key	Description
gapi	Version of the generator configuration file API. Only 1.0 is defined
files_root	Directory where input files are found. FuseSoC sets this to the calling core's file directory
vlnv	Colon-separated VLNV identifier to use for the output core. A generator is free to ignore this and use another VLNV.
parameters	Everything contained under the parameters key should be treated as instance-specific configuration for the generator

Example yaml configuration file:

```
files_root: /home/user/cores/mysoc
gapi: '1.0'
parameters:
  masters:
    dbus:
      slaves: [sdram_dbus, uart0, gpio0, gpio1, spi0]
    or1k_i:
      slaves: [sdram_ibus, rom0]
  slaves:
    gpio0: {datawidth: 8, offset: 2432696320, size: 2}
    gpio1: {datawidth: 8, offset: 2449473536, size: 2}
    rom0: {offset: 4026531840, size: 1024}
```

(continues on next page)

(continued from previous page)

```
    sdram_dbus: {offset: 0, size: 33554432}
    sdram_ibus: {offset: 0, size: 33554432}
    spi0: {datawidth: 8, offset: 2952790016, size: 8}
    uart0: {datawidth: 8, offset: 2415919104, size: 32}
vlnv: ::mysoc-wb_intercon:0
```

The above example is for a generator that creates verilog code for a wishbone interconnect.

Registering a generator

When FuseSoC scans the flattened dependency tree of a core, it will look for a section called *generators* in each core file. This section is used by cores to notify FuseSoC that they contain a generator and describe how to use it.

The generators section contain a map of generator sections so that each core is free to define multiple generators. The key of each generator decide its name

The following keys are valid in a generator section.

command: The command to run (relative to the core root) to invoke the generator. FuseSoC will pass a yaml configuration file as the first argument when calling the command. interpreter: If the command requires an interpreter (e.g. python or perl), this will be used called, with the string specified in *command* as the first argument, and the yaml file as the second argument.

Example generator section from a CAPI2 core file

```
generators:
  wb_intercon_gen:
    interpreter: python
    command: sw/wb_intercon_gen
```

The above snippet will register a generator with the name wb_intercon_gen. This name will be used by cores that wish to invoke the generator. When the generator is invoked it will run *python /path/to/core/sw/wb_intercon_gen* from the sw subdirectory of the core where the generators section is defined.

Calling a generator

The final piece of the generators machinery is to run a generator with some specific parameters. This is done by creating a special section in the core that wishes to use a generator and adding this section to the targets that need it. Using the same example generator as previously, this section could look like the example below:

```
generate:
  wb_intercon:
    generator : wb_intercon_gen
    parameters:
      masters:
        or1k_i:
          slaves:
            - sdram_ibus
            - rom0
        dbus:
          slaves: [sdram_dbus, uart0, gpio0, gpio1, spi0]

      slaves:
        sdram_dbus:
          offset : 0
```

(continues on next page)

(continued from previous page)

```
      size : 0x2000000

    sdram_ibus:
      offset: 0
      size: 0x2000000

    uart0:
      datawidth: 8
      offset: 0x90000000
      size: 32

    gpio0:
      datawidth: 8
      offset: 0x91000000
      size: 2

    gpio1:
      datawidth: 8
      offset: 0x92000000
      size: 2

    spi0:
      datawidth: 8
      offset: 0xb0000000
      size: 8

    rom0:
      offset: 0xf0000000
      size: 1024
```

The above core file snippet will register a parametrized generator instance with the name wb_intercon. It will use the generator called *wb_intercon_gen* which FuseSoC has previously found in the depedency tree. Everything listed under the *parameters* key is instance-specific configuration to be sent to the generator.

Just registering a generate section will not cause the generator to be invoked. It must also be listed in the target and the generator to be used must be in the dependency tree. The following snippet adds the parameterized generator to the *default* target and adds an explicit dependency on the core that contains the generator. As CAPI2 cores only allow filesets to have dependencies, an empty fileset for this purpose must be created

```
filesets:
  wb_intercon_dep:
    depend:
      [wb_intercon]

targets:
  default:
    filesets : [wb_intercon_dep]
    generate : [wb_intercon]
```

When FuseSoC is launched and a core target using a generator is processed, the following will happen for each entry in the target's *generate* entry.

1. A key lookup is performed in the core file's *generate* section to find the generator configuration
2. FuseSoC checks that it has registered a generator by the name specified in the *generator* entry of the configuration.
3. FuseSoC calculates a unique VLNV for the generator instance by taking the calling core's VLNV and concati-

nating the name field with the generator instance name.

4. A directory is created under <cache_root>/generated with a sanitized version of the calculated VLNV. This directory is where the output from the generator eventually will appear.
5. A yaml configuration file is created in the generator output directory. The parameters from the instance are passed on to this file. FuseSoC will set the files root of the calling core as *files_root* and add the calculated vlnv.
6. FuseSoC will switch working directory to the generator output directory and call the generator, using the command found in the generator's *command* field and with the created yaml file as command-line argument.
7. When the generator has successfully completed, FuseSoC will scan the generator output directory for new .core files. These will be injected in the dependency tree right after the calling core and will be treated just like regular cores, except that any extra dependencies listed in the generated core will be ignored.

1.5.6 Hooks: intercept the build process

Todo: Document hooks.

1.5.7 VPI Support

Todo: Document VPI support.

1.6 The FuseSoC package manager

Todo: This section of the documentation is copied over from previous documentation and needs to be edited in style and extended.

1.6.1 Core libraries

A collection of one or more cores in a directory tree is called a core library. FuseSoC supports working with multiple core libraries. The locations of the libraries are specified in the FuseSoC configuration file, `fusesoc.conf`

To find a configuration file, FuseSoC will first look for `fusesoc.conf` in the current directory, and if there is no file there, it will search next in `$XDG_CONFIG_HOME/fusesoc` (i.e. `~/.config/fusesoc` on Linux and `%HOMEPATH%\.config\fusesoc` on Windows) and lastly in `/etc/fusesoc`

By running `fusesoc init` after FuseSoC is installed, the standard libraries will be installed, and a default configuration file will be created in `$XDG_CONFIG_HOME/fusesoc/fusesoc.conf` on Linux and `%HOMEPATH%\.config\fusesoc\fusesoc.conf` on Windows with the following contents:

```
[library.orpsoc-cores]
sync-uri = https://github.com/openrisc/orpsoc-cores
sync-type = git

[library.fusesoc-cores]
```

(continues on next page)

(continued from previous page)

```
sync-uri = https://github.com/fusesoc/fusesoc-cores
sync-type = git
```

1.6.2 Core search order

Once FuseSoC has found its configuration file, it will parse the `cores_root` option in the `[main]` section of `fusesoc.conf`. This option is a space-separated list of library locations which are searched in the order they appear. Additional library locations can be added on the command line by setting the `--cores-root` parameter when FuseSoC is launched. The library locations specified from the command-line will be parsed after those in `fusesoc.conf`

For each library location, FuseSoC will recursively search for files with a *.core* suffix. Each of these files will be parsed and addded to the in-memory FuseSoC database if they are valid `.core` files.

Several `.core` files can reside in the same directory and they will all be parsed.

If several cores with the same VLNV identifier are encountered the latter will replace the former. This can be used to override cores in a library with an alternative core in another library by specifying them in a library that will be parsed later, either temporarily by adding `--cores-root` to the command-line, or permanently by adding the other library at the end of fusesoc.conf

1.7 Common Problems and Solutions

1.7.1 Making changes to cores in a library

A common situation is that a user wants to use their own copy of a core, instead of the one provided by a library, for example to fix a bug or add new functionality. The following steps can be used to achieve this:

Example. Replace a core in a library with a user-specified version

1. Create a new directory to keep the user-copies of the cores (this directory will be referred to as `$corelib` from now on)
2. Download the core source (the repository or URL can be found in the `[provider]` section of the original core)
3. *If the downloaded core already contains a .core file, this step is ignored* Copy the original .core file to the root of the downloaded core. Edit the file and remove the `[provider]` section. (This will stop FuseSoC from downloading the core and use files from the directory containing the .core file instead)
4. Add `$corelib` to the end of your library search path, either by editing `fusesoc.conf` or by adding `--cores-root=$corelib` to the command-line arguments
5. Verify that the new core is found by running fusesoc core-info $core. Check the output to see that "Core root:" is set to the directory where the core was downloaded

CHAPTER

TWO

FUSESOC REFERENCE MANUAL

2.1 CAPI2 Reference

CAPI2 (Core API version 2) describes the properties of a core as a YAML data structure.

2.1.1 Types

File

A File object represents a physical file. It can be a simple string, with the path to the file relative to the core root (e.g. *path/to/file.v*). It is also possible to assign attributes to a file, by using the file name as a dictionary key and the attributes as a map. (e.g. *path/to/file.v : {is_include_file : true, file_type : systemVerilogSource}*). Valid attributes are

Attribute	Type	Description
is_include_file	bool	Treats file as an include file when true
include_path	str	Explicitly set an include directory, relative to core root, instead of the directory containing the file
file_type	str	File type. Overrides the file_type set on the containing fileset
logical_name	str	Logical name, i.e. library for VHDL/SystemVerilog. Overrides the logical_name set on the containing fileset

Genparams

Genparams are private configuration for a generator. Normally specified as a map of key/value pairs

Provider

Specifies how to fetch the core. The presence of a provider section indicates this is a remote core that has its source code separated from the core description file.

String

String is an unparsed (plain) string.

StringWithUseFlags

StringWithUseFlags is a string that can contain CAPI2 expressions that are evaluated during parsing.

CAPI2 expressions are used to evaluate an expression only if a flag is set or unset. The general form is *flag_is_set ? (expression)* to evaluate *expression* if flag is set or *!flag_is_set ? (expression)* to evaluate *expression* if flag is not set.

Example Only include fileset *verilator_tb* when the target is used with verilator

```
filesets :  [rtl, tb, tool_verilator? (verilator_tb)]
```

StringWithUseFlagsOrDict

Item is allowed to be either a *StringWithUseFlags* or a dict of *StringWithUseFlags*.

StringWithUseFlagsOrList

Item is allowed to be either a *StringWithUseFlags* or a list of *StringWithUseFlags*

VInv

:-separated VLNV (Vendor, Library, Name, Version) identifier

Dict

A dictionary.

Bool

A boolean.

Integer

An integer.

List

All keys that are defined as lists have a corresponding <*key*>*_append* key. The items in the <*key*>*_append* list is automatically appended to the list of the original key.

Example A target contains the following keys

```
filesets : [fileset_a, fileset_b]
filesets_append : [fileset_c, fileset_d]
```

This will be evaluated to

```
filesets : [fileset_a, fileset_b, fileset_c, fileset_d]
```

The main use case for this feature is inheritance between targets with yaml anchors as in the following example where a board-specific implementation and a simulation target inherits from the default target

```
targets:
  default: &base
    filesets: [core]

  myboard:
    <<: *base
    filesets_append: [myboard_files]

  sim:
    <<: *base
    filesets_append: [testbench]
```

2.1.2 Sections

The first table lists all valid keywords in the document root while the other tables are keywords for sub-sections of the tree

Root elements of the CAPI2 structure

Field Name	Type	Description
name	*Vlnv*	VLNV identifier for core
description	*String*	Short description of core
provider	*Provider*	Provider of core
CAPI=2	*String*	Technically a header. Must appear as the first line in the core description file
filesets	Dict of *Fileset*	File sets
generate	Dict of *Generate*	Parametrized generator configurations
generators	Dict of *Generators*	Generator provided by this core
scripts	Dict of *Script*	Scripts that are used by the hooks
targets	Dict of *Target*	Available targets
parameters	Dict of *Parameter*	Available parameters
vpi	Dict of *Vpi*	Available VPI modules

Fileset

A fileset represents a group of file with a common purpose. Each file in the fileset is required to have a file type and is allowed to have a logical_name which can be set for the whole fileset or individually for each file. A fileset can also have dependencies on other cores, specified in the depend section

Field Name	Type	Description
file_type	*String*	Default file_type for files in fileset
logical_name	*String*	Default logical_name (i.e. library) for files in fileset
files	List of *File*	Files in fileset
depend	List of *StringWithUseFlags*	Dependencies of fileset

Generate

The elements in this section each describe a parameterized instance of a generator. They specify which generator to invoke and any generator-specific parameters.

Field Name	Type	Description
generator	*String*	The generator to use. Note that the generator must be present in the dependencies of the core.
parameters	*Generator params*	Generator-specific parameters. `fusesoc gen show $generator` might show available parameters
position	*String*	Where to insert the generated core. Legal values are *first*, *append* or *last*. *append* will insert core after the core that called the generator

Generators

Generators are custom programs that generate FuseSoC cores. They are generally used during the build process, but can be used stand-alone too. This section allows a core to register a generator that can be used by other cores.

Field Name	Type	Description
command	*String*	The command to run (relative to the core root)
interpreter	*String*	If the command needs a custom interpreter (such as python) this will be inserted as the first argument before command when calling the generator. The interpreter needs to be on the system PATH.
description	*String*	Short description of the generator, as shown with `fusesoc gen list`
usage	*String*	A longer description of how to use the generator, including which parameters it uses (as shown with `fusesoc gen show $generator`).

Target

A target is the entry point to a core. It describes a single use-case and what resources that are needed from the core such as file sets, generators, parameters and specific tool options. A core can have multiple targets, e.g. for simulation, synthesis or when used as a dependency for another core. When a core is used, only a single target is active. The *default* target is a special target that is always used when the core is being used as a dependency for another core or when no `--target=` flag is set.

Field Name	Type	Description
default_tool	*String*	Default tool to use unless overridden with `--tool=`
description	*String*	Description of the target
hooks	*Hooks*	Script hooks to run when target is used
tools	*Tools*	Tool-specific options for target
toplevel	*StringWithUseFlagsOrList*	Top-level module. Normally a single module/entity but can be a list of several items
filesets	List of *StringWithUseFlags*	File sets to use in target
generate	List of *StringWithUseFlagsOrDict*	Parameterized generators to run for this target with optional parametrization
parameters	List of *StringWithUseFlags*	Parameters to use in target. The parameter default value can be set here with `param=value`
vpi	List of *StringWithUseFlags*	VPI modules to build and include for target

Tools

The valid subsections of the Tools section and their options are defined by what Edalize backends are available at runtime. The sections listed here are the ones that were available when the documentation was generated.

Field Name	Type	Description
ascentlint	*Ascentlint*	Ascentlint-specific options
diamond	*Diamond*	Diamond-specific options
ghdl	*Ghdl*	Ghdl-specific options
icarus	*Icarus*	Icarus-specific options
icestorm	*Icestorm*	Icestorm-specific options
ise	*Ise*	Ise-specific options
isim	*Isim*	Isim-specific options
modelsim	*Modelsim*	Modelsim-specific options
morty	*Morty*	Morty-specific options
quartus	*Quartus*	Quartus-specific options
radiant	*Radiant*	Radiant-specific options
rivierapro	*Rivierapro*	Rivierapro-specific options
spyglass	*Spyglass*	Spyglass-specific options
symbiflow	*Symbiflow*	Symbiflow-specific options
symbiyosys	*Symbiyosys*	Symbiyosys-specific options
trellis	*Trellis*	Trellis-specific options
vcs	*Vcs*	Vcs-specific options
veribleformat	*Veribleformat*	Veribleformat-specific options
veriblelint	*Veriblelint*	Veriblelint-specific options
verilator	*Verilator*	Verilator-specific options
vivado	*Vivado*	Vivado-specific options
vunit	*Vunit*	Vunit-specific options
xcelium	*Xcelium*	Xcelium-specific options
xsim	*Xsim*	Xsim-specific options
yosys	*Yosys*	Yosys-specific options

Hooks

Hooks are scripts that are run at different points in the build process. They are always launched from the work root

Field Name	Type	Description
pre_build	List of *StringWithUseFlags*	Scripts executed before the *build* phase
post_build	List of *StringWithUseFlags*	Scripts executed after the *build* phase
pre_run	List of *StringWithUseFlags*	Scrips executed before the *run* phase
post_run	List of *StringWithUseFlags*	Scripts executed after the *run* phase

Parameter

A parameter is a compile-time or run-time configuration of a core.

Field Name	Type	Description
datatype	*String*	Parameter datatype. Legal values are *bool*, *file*, *int*, *str*. *file* is same as *str*, but prefixed with the current directory that FuseSoC runs from
default	*String*	Default value
description	*String*	Description of the parameter, as can be seen with `fusesoc run --target=$target $core --help`
paramtype	*StringWithUseFlags*	Specifies type of parameter. Legal values are *cmdlinearg* for command-line arguments directly added when running the core, *generic* for VHDL generics, *plusarg* for verilog plusargs, *vlogdefine* for Verilog `` *define*` or *vlogparam* for verilog top-level parameters. All paramtypes are not valid for every backend. Consult the backend documentation for details.
scope	*String*	**Not used** : Kept for backwards compatibility

Script

A script specifies how to run an external command that is called by the hooks section together with the actual files needed to run the script. Scripts are alway executed from the work root

Field Name	Type	Description
env	Dict of *String*	Map of environment variables to set before launching the script
cmd	List of *String*	List of command-line arguments
filesets	List of *String*	Filesets needed to run the script

Vpi

A VPI (Verilog Procedural Interface) library is a shared object that is built and loaded by a simulator to provide extra Verilog system calls. This section describes what files and external libraries to use for building a VPI library

Field Name	Type	Description
libs	List of *String*	External libraries to link against
filesets	List of *String*	Filesets containing files to use when compiling the VPI library

Ascentlint

Real Intent Ascent Lint backend

Ascent Lint performs static source code analysis on HDL code and checks for common coding errors or coding style violations.

Field Name	Type	Description
ascentlint_options	List of *String*	Additional run options for ascentlint

Diamond

Backend for Lattice Diamond

Field Name	Type	Description
part	*String*	FPGA part number (e.g. LFE5U-45F-6BG381C)

Ghdl

GHDL is an open source VHDL simulator, which fully supports IEEE 1076-1987, IEEE 1076-1993, IEE 1076-2002 and partially the 1076-2008 version of VHDL

Field Name	Type	Description
analyze_options	List of *String*	Options to use for the import (ghdl -i) and make (ghdl -m) phases
run_options	List of *String*	Options to use for the run (ghdl -r) phase

Icarus

Icarus Verilog is a Verilog simulation and synthesis tool. It operates as a compiler, compiling source code written in Verilog (IEEE-1364) into some target format

Field Name	Type	Description
timescale	*String*	Default timescale
iverilog_options	List of *String*	Additional options for iverilog

Icestorm

Open source toolchain for Lattice iCE40 FPGAs. Uses yosys for synthesis and arachne-pnr or nextpnr for Place & Route

Field Name	Type	Description
pnr	*String*	Select Place & Route tool. Legal values are *arachne* for Arachne-PNR or *next* for nextpnr. Default is arachne
arch	*String*	Target architecture. Legal values are *xilinx*, *ice40* and *ecp5*
output_format	*String*	Output file format. Legal values are *json*, *edif*, *blif*
yosys_as_subtool	*bool*	Determines if Yosys is run as a part of bigger toolchain, or as a standalone tool
makefile_name	*String*	Generated makefile name, defaults to $name.mk
script_name	*String*	Generated tcl script filename, defaults to $name.mk
arachne_pnr_options	List of *String*	Additional options for Arachnhe PNR
nextpnr_options	List of *String*	Additional options for nextpnr
yosys_synth_options	List of *String*	Additional options for the synth_ice40 command

Ise

Xilinx ISE Design Suite

Field Name	Type	Description
family	*String*	FPGA family (e.g. spartan6)
device	*String*	FPGA device (e.g. xc6slx45)
package	*String*	FPGA package (e.g. csg324)
speed	*String*	FPGA speed grade (e.g. -2)

Isim

Xilinx ISim simulator from ISE design suite

Field Name	Type	Description
fuse_options	List of *String*	Additional options for compilation with FUSE
isim_options	List of *String*	Additional run options for ISim

Modelsim

ModelSim simulator from Mentor Graphics

Field Name	Type	Description
vcom_options	List of *String*	Additional options for compilation with vcom
vlog_options	List of *String*	Additional options for compilation with vlog
vsim_options	List of *String*	Additional run options for vsim

Morty

Run the (System-) Verilog pickle tool called *morty*.

Field Name	Type	Description
morty_options	List of *String*	Run-time options passed to morty.

Quartus

The Quartus backend supports Intel Quartus Std and Pro editions to build systems and program the FPGA

Field Name	Type	Description
family	*String*	FPGA family (e.g. Cyclone V)
device	*String*	FPGA device (e.g. 5CSXFC6D6F31C8ES)
cable	*String*	Specifies the FPGA's JTAG programming cable. Use the tool *jtagconfig* to determine the available cables.
board_device_index	*String*	Specifies the FPGA's device number in the JTAG chain. The device index specifies the device where the flash programmer looks for the Nios® II JTAG debug module. JTAG devices are numbered relative to the JTAG chain, starting at 1. Use the tool *jtagconfig* to determine the index.
pnr	*String*	P&R tool. Allowed values are quartus (default), dse (to run Design Space Explorer) and none (to just run synthesis)
dse_options	List of *String*	Options for DSE (Design Space Explorer)
quartus_options	List of *String*	Additional options for Quartus

Radiant

Backend for Lattice Radiant

Field Name	Type	Description
part	*String*	FPGA part number (e.g. LIFCL-40-9BG400C)

Rivierapro

Riviera Pro simulator from Aldec

Field Name	Type	Description
compilation_mode	*String*	Common or separate compilation, sep - for separate compilation, common - for common compilation
vlog_options	List of *String*	Additional options for compilation with vlog
vsim_options	List of *String*	Additional run options for vsim

Spyglass

Synopsys (formerly Atrenta) Spyglass Backend

Spyglass performs static source code analysis on HDL code and checks for common coding errors or coding style violations.

Example snippet of a CAPI2 description file

```
spyglass:
  methodology: "GuideWare/latest/block/rtl_handoff"
  goals:
    - lint/lint_rtl
  spyglass_options:
    # prevent error SYNTH_5273 on generic RAM descriptions
    - handlememory yes
  rule_parameters:
    # Allow localparam to be used in case labels (e.g. in state machines)
    - handle_static_caselabels yes
```

Field Name	Type	Description
methodology	*String*	
goals	List of *String*	
spyglass_options	List of *String*	
rule_parameters	List of *String*	

Symbiflow

The Symbiflow backend executes Yosys sythesis tool and VPR place and route. It can target multiple different FPGA vendors

Field Name	Type	Description
package	*String*	FPGA chip package (e.g. clg400-1)
part	*String*	FPGA part type (e.g. xc7a50t)
vendor	*String*	Target architecture. Currently only "xilinx" is supported
pnr	*String*	Place and Route tool. Currently only "vpr" is supported
vpr_options	*String*	Additional vpr tool options. If not used, default options for the tool will be used
environment_script	*String*	Optional bash script that will be sourced before each build step.

Symbiyosys

SymbiYosys formal verification wrapper for Yosys

Field Name	Type	Description
tasknames	List of *String*	A list of the .sby file's tasks to run. Passed on the sby command line.

Trellis

Project Trellis enables a fully open-source flow for ECP5 FPGAs using Yosys for Verilog synthesis and nextpnr for place and route

Field Name	Type	Description
arch	*String*	Target architecture. Legal values are *xilinx*, *ice40* and *ecp5*
output_format	*String*	Output file format. Legal values are *json*, *edif*, *blif*
yosys_as_subtool	*bool*	Determines if Yosys is run as a part of bigger toolchain, or as a standalone tool
makefile_name	*String*	Generated makefile name, defaults to $name.mk
script_name	*String*	Generated tcl script filename, defaults to $name.mk
nextpnr_options	List of *String*	Additional options for nextpnr
yosys_synth_options	List of *String*	Additional options for the synth_ecp5 command

Vcs

Synopsys VCS Backend

VCS is one of the "Big 3" simulators.

Example snippet of a CAPI2 description file for VCS:

```
vcs:
  vcs_options:
    # Compile-time options passed to the vcs command
    - -debug_access+pp
    - -debug_access+all
  run_options:
    # Run-time options passed to the simulation itself
    - -licqueue
```

Field Name	Type	Description
vcs_options	List of *String*	
run_options	List of *String*	

Veribleformat

Verible format backend (verible-verilog-format)

Field Name	Type	Description
verible_format_args	List of *String*	Extra command line arguments passed to the Verible tool

Veriblelint

Verible lint backend (verible-verilog-lint)

Field Name	Type	Description
ruleset	*String*	Ruleset: [default|all|none]
verible_lint_args	List of *String*	Extra command line arguments passed to the Verible tool
rules	List of *String*	What rules to use. Prefix a rule name with "-" to disable it.

Verilator

Verilator is the fastest free Verilog HDL simulator, and outperforms most commercial simulators

Field Name	Type	Description
mode	*String*	Select compilation mode. Legal values are *cc* for C++ testbenches, *sc* for SystemC testbenches or *lint-only* to only perform linting on the Verilog code
cli_parser	*String*	**Deprecated: Use run_options instead** : Select whether FuseSoC should handle command-line arguments (*managed*) or if they should be passed directly to the verilated model (*raw*). Default is *managed*
libs	List of *String*	Extra libraries for the verilated model to link against
verilator_options	List of *String*	Additional options for verilator
make_options	List of *String*	Additional arguments passed to make when compiling the simulation. This is commonly used to set OPT/OPT_FAST/OPT_SLOW.
run_options	List of *String*	Additional arguments directly passed to the verilated model

Vivado

The Vivado backend executes Xilinx Vivado to build systems and program the FPGA

Field Name	Type	Description
vivado_settings	*String*	Path to vivado settings (e.g. /opt/Xilinx/Vivado/2017.2/settings64.sh)
part	*String*	FPGA part number (e.g. xc7a35tcsg324-1)
synth	*String*	Synthesis tool. Allowed values are vivado (default) and yosys.
pnr	*String*	P&R tool. Allowed values are vivado (default) and none (to just run synthesis)
jtag_freq	*Integer*	The frequency for jtag communication
hw_target	**`Description`_**	Board identifier (e.g. */xilinx_tcf/Digilent/123456789123A

Vunit

VUnit testing framework

Field Name	Type	Description
vunit_runner	*String*	Name of the Python file exporting a "VUnitRunner" class that is used to configure and execute test
add_libraries	List of *String*	A list of framework libraries to add. Allowed values include "array_util", "com", "json4hdl", "osvvm", "random", "verification_components"
vunit_options	List of *String*	Options to pass to the VUnit test runner

Xcelium

Xcelium simulator from Cadence Design Systems

Field Name	Type	Description
xmvhdl_options	List of *String*	Additional options for compilation with xmvhdl
xmvlog_options	List of *String*	Additional options for compilation with xmvlog
xmsim_options	List of *String*	Additional run options for xmsim
xrun_options	List of *String*	Additional run options for xrun

Xsim

XSim simulator from the Xilinx Vivado suite

Field Name	Type	Description
xelab_options	List of *String*	Additional options for compilation with xelab
xsim_options	List of *String*	Additional run options for XSim

Yosys

Open source synthesis tool targeting many different FPGAs

Field Name	Type	Description
arch	*String*	Target architecture. Legal values are *xilinx*, *ice40* and *ecp5*
output_format	*String*	Output file format. Legal values are *json*, *edif*, *blif*
yosys_as_subtool	*bool*	Determines if Yosys is run as a part of bigger toolchain, or as a standalone tool
makefile_name	*String*	Generated makefile name, defaults to $name.mk
script_name	*String*	Generated tcl script filename, defaults to $name.mk
yosys_synth_options	List of *String*	Additional options for the synth command

2.2 CAPI1 (deprecated)

2.2.1 Type definitions

File

File objects consist of a mandatory file name, with path relative to the core root. Extra options can be specified as a comma-separated list enclosed in `[]` after the file name. Options are either boolean (`option`) or has a value (`option=value`). No white-space is allowed anywhere in the file object.

The following options are defined:

- **file_type :** Value can be any type defined in *File types*
- **is_include_file :** Boolean value to indicate this should be treated as an include file
- **logical_name :** Indicate that the file belongs to a logical unit (e.g. VHDL Library) with the name set by the value
- **copyto :** Indicate that the file should be copied to a new location relative to the work root.

Example: `rtl/verilog/uart_defines.v[file_type=verilogSource,is_include_file]`

Example: `data/mem_init_file.bin[copyto=out/boot.bin]`

FileList

Space-separated list of *File*

Each element in the list is first subjected to the expansion according to *PathList* and then parsed as a *File*

PathList

Space-separated list of paths

Each element in the list is subjected to expansion of environment variables and ~ to home directories

SimulatorList

List of supported simulators. Allowed values are ghdl, icarus, isim, modelsim, vcs, verilator, xsim

SourceType

Language used for Verilator testbenches. Allowed values are C, CPP or systemC

StringList

Space-separated list of strings

VlnvList

Space-separated list of VLNV tags

Each element is treated as a VLNV element with an optional version range

Example:

```
librecores.org:peripherals:uart16550:1.5 >=::simple_spi:1.6
mor1kx =::i2c:1.14``
```

2.2.2 File types

The following valid file types are defined: PCF, QIP, SDC, UCF, BMM, tclSource, user, verilogSource, verilogSource-95, verilogSource-2001, verilogSource-2005, systemVerilogSource, systemVerilogSource-3.0, systemVerilogSource-3.1, systemVerilogSource-3.1a, vhdlSource, vhdlSource-87, vhdlSource-93, vhdlSource-2008, xci, xdc

2.2.3 Sections

fileset

Name	Type	Description
file_type	String	Default file type of the files in fileset
files	*FileList*	List of files in fileset
is_include_file	String	Specify all files in fileset as include files
logical_name	String	Default logical_name (e.g. library) of the files in fileset
scope	String	Visibility of fileset (private/public). Private filesets are only visible when this core is the top-level. Public filesets are visible also for cores that depend on this core. Default is public
usage	*StringList*	List of tags describing when this fileset should be used. Can be general such as sim or synth, or tool-specific such as quartus, verilator, icarus. Defaults to *sim synth*.

ghdl

Name	Type	Description
analyze_options	*StringList*	Extra GHDL analyzer options
depend	*VlnvList*	Tool-specific Dependencies
run_options	*StringList*	Extra GHDL run options

icarus

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
iverilog_options	*StringList*	Extra Icarus verilog compile options

icestorm

Name	Type	Description
arachne_pnr_options	*StringList*	arachne-pnr options
depend	*VlnvList*	Tool-specific Dependencies
pcf_file	*FileList*	Physical constraint file
top_module	String	RTL top-level module
yosys_synth_options	*StringList*	Additional options for the `synth_*` commands in yosys

ise

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
device	String	FPGA device identifier
family	String	FPGA device family
package	String	FPGA device package
speed	String	FPGA device speed grade
tcl_files	*FileList*	Extra TCL scripts
top_module	String	RTL top-level module
ucf_files	*FileList*	UCF constraint files

isim

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
isim_options	*StringList*	Extra Isim compile options

main

Name	Type	Description
backend	String	Backend for FPGA implementation
component	*PathL ist*	Core IP-Xact component file
depend	*VlnvList*	Common dependencies
description	String	Core description
name	String	Component name
patches	*StringList*	FuseSoC-specific patches
simulators	*SimulatorList*	Supported simulators. Valid values are icarus, modelsim, verilator, isim and xsim. Each simulator have a dedicated section desribed elsewhere in this document

modelsim

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
vlog_options	*StringList*	Additional arguments for vlog
vsim_options	*StringList*	Additional arguments for vsim

parameter

Name	Type	Description
datatype	String	Data type of argument (int, str, bool, file
default	String	Default value of argument
description	String	Parameter description
paramtype	String	Type of parameter (plusarg, vlogparam, generic, cmdlinearg
scope	String	Visibility of parameter. Private parameters are only visible when this core is the top-level. Public parameters are visible also when this core is pulled in as a dependency of another core

quartus

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
device	String	FPGA device identifier
family	String	FPGA device family
qsys_files	*FileList*	Qsys IP description files
quartus_options	String	Quartus command-line options
sdc_files	*FileList*	SDC constraint files
tcl_files	*FileList*	Extra script files
top_module	String	RTL top-level module

rivierapro

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
vlog_options	*StringList*	Additional arguments for vlog
vsim_options	*StringList*	Additional arguments for vsim

scripts

Name	Type	Description
post_impl_scripts	*StringList*	Scripts to run after backend implementation
post_run_scripts	*StringList*	Scripts to run after simulations
pre_build_scripts	*StringList*	Scripts to run before building
pre_run_scripts	*StringList*	Scripts to run before running simulations
pre_synth_scripts	*StringList*	Scripts to run before backend synthesis

trellis

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
nextpnr_options	*StringList*	nextpnr options
top_module	String	RTL top-level module
yosys_synth_options	*StringList*	Additional options for the `synth_*` commands in yosys

vcs

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
vcs_options	*StringList*	Extra vcs compile options

verilator

Name	Type	Description
cli_parser	String	Select CLI argument parser. Set to *fusesoc* to handle parameter sections like other simulators. Set to *passthrough* to send the arguments directly to the verilated model. Default is *passthrough*
define_files	*PathL ist*	Verilog include files containing `define directives to be converted to C #define directives in corresponding .h files (deprecated)
depend	*VlnvList*	Tool-specific Dependencies
include_files	*FileList*	Verilator testbench C include files
libs	*PathL ist*	External libraries linked with the generated model
source_type	String	Testbench source code language (Legal values are systemC, C, CPP. Default is C)
src_files	*FileList*	Verilator testbench C/cpp/sysC source files
tb_toplevel	*FileList*	Testbench top-level C/C++/SC file
top_module	String	verilog top-level module
verilator_options	*StringList*	Verilator build options

verilog

Name	Type	Description
file_type	String	Default file type of the files in fileset
include_files	*FileList*	Verilog include files
src_files	*FileList*	Verilog source files for synthesis/simulation
tb_include_files	*FileList*	Testbench include files
tb_private_src_files	*FileList*	Verilog source files that are only used in the core's own testbench. Not visible to other cores
tb_src_files	*FileList*	Verilog source files that are only used in simulation. Visible to other cores

vhdl

Name	Type	Description
src_files	*PathL ist*	VHDL source files for simulation and synthesis

vivado

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
hw_device	String	FPGA device identifier
part	String	FPGA device part
top_module	String	RTL top-level module

vpi

Name	Type	Description
include_files	*FileList*	C include files for VPI library
libs	*StringList*	External libraries linked with the VPI library
src_files	*FileList*	C source files for VPI library

xsim

Name	Type	Description
depend	*VlnvList*	Tool-specific Dependencies
xsim_options	*StringList*	Extra Xsim compile options

provider

The provider section gives information on where to find the source code for the core. If the provider section is missing, the core is assumed to be local, with the directory of the .core file as the root directory.

Name	Type	Description
name	String	The name option selects which provider backend to use. All other provider options are specific to the selected provider. Currently supported backends are github, git, opencores, submodule and url.
cachable	boolean	If the cachable option is set to false, FuseSoc will unconditionally refetch the core even if it is found in the cache. Default is true

Provider-specific options:

github

- **user :** Name of the github user or organisation.
- **repo :** Name of the GIT repository.
- **version :** Name of the GIT ref (i.e. commit SHA, branch or tag) to use

git

- **repo :** URL of the GIT repository.
- **version :** Name of the GIT ref (i.e. commit SHA, branch or tag) to use

opencores

- **repo_name :** Name of the opencores project. Can be found under Details on the project homepage.
- **repo_root :** The sub directory in the repo that contains the files of interest. In most cases the value "trunk" is used to avoid pulling in tags and branches.
- **revision :** The svn revision of the repository.

url

- **url :** URL of the core file (or archive).
- **filetype :** File type (zip, tar, simple).

2.2.4 Known issues

1. Spaces are not allowed anywhere in the paths.

2.3 Migration guide

FuseSoC strives to be backwards-compatible, but as new features are added to FuseSoC, some older features become obsolete. This chapter contains information on how to migrate away from deprecated features to keep the core description files up-to-date with the latest best practices.

2.3.1 Migrating from CAPI1 to CAPI2

Why

FuseSoC's *.core* files are written in a "language" called CAPI. The current version of CAPI is version 2, also called CAPI2. Going forward, only the newer CAPI2 file format will be supported, which simplifies the use and implementation of FuseSoC greatly.

When

In FuseSoC 1.x, both CAPI1 and CAPI2 are supported. Starting with FuseSoC 2 only CAPI2 will be supported. To be able to update to the next version of FuseSoC seamlessly you need to migrate your existing CAPI1 core files to CAPI2. We recommend doing this migration *now* while still running FuseSoC 1.x.

How

FuseSoC ships with an automated conversion tool from CAPI1 to CAPI2, which provides a solid starting point for the conversion process. However, even though CAPI2 supports almost all features of CAPI1, there isn't always a 1:1 mapping between the two formats. Therefore the automatic conversion won't be always correct, and a bit of manual cleanup work will be needed.

To convert a single core file, follow these steps:

1. Ensure you're running the latest version of FuseSoC 1.x.
2. Ensure that you have a backup of your core file, or have committed the current version in a version control system.
3. Run `fusesoc migrate-capi1-to-capi2 --inplace your_core_file.core` to convert the file automatically.
4. Open `your_core_file.core` to check the conversion output and adjust it as necessary (e.g. add back comments, adjust the ordering of statements, etc.)

You can find documentation on the *CAPI1* as well as on *CAPI2* in the reference manual. We also recommend to have a look at the *Writing core files* section in the user guide for current best practices on writing CAPI2 core files.

To convert all core files in a directory, Linux users can run the following command:

```
find your_coredir -iname '*.core' -exec fusesoc migrate-capi1-to-capi2 --nowarn --
↪inplace {} \;
```

If you get stuck in the conversion process, or if a CAPI1 feature you rely on isn't available in CAPI2, please get in touch by filing an issue on GitHub.

2.3.2 Migrating from .system files

Why

The synthesis backends required a separate .system file in addition to the .core file. There is however very little information in the .system file, it was never properly documented and some information is duplicated from the .core file. For these reasons a decision was made to drop the .system file and move the relevant information to the .core file instead.

When

`.system` files are no longer needed as of FuseSoC 1.6

The `.system` file will still be supported for some time to allow users to perform the migration, but any equivalent options in the `.core` file will override the ones in `.system`

How

Perform the following steps to migrate from .system files

1. Move the `backend` parameter from the `main` section in the `.system` file to the `main` section in the `.core` file
2. Move the backend section (i.e. `icestorm`, `ise`, `quartus` or `vivado`) to the `.core` file
3. Move `pre_build_scripts` from the `scripts` section in the `.system` file to `pre_synth_scripts` in the `scripts` section in the `.core` file.
4. Move `post_build_scripts` from the `scripts` section in the `.system` file to `post_impl_scripts` in the `scripts` section in the `.core` file.

2.3.3 Migrating from plusargs

Why

Up until FuseSoC 1.3, verilog plusargs were the only way to set external run-time parameters. Cores could register which plusargs they supported through the `plusargs` section. This mechanism turned out to be too limited, and in order to support public/private parameters, defines, VHDL generics etc, `parameter` sections were introduced to replace the `plusargs` section.

When

`parameter` sections were introduced in FuseSoC 1.3

The `plusargs` section is still supported to allow time for migrations

How

Entries in the `plusargs` section are described as `<name> = <type> <description>`. For each of these entries, create a new section with the following contents

```
[parameter <name>]
datatype = <type>
description = <description>
paramtype = plusarg
```

The `parameter` sections also support the additional tags `default`, to set a default value, and `scope` to select if this parameter should be visible to other cores (`scope=public`) or only when this core is used as the toplevel (`scope=private`).

2.3.4 Migrating to filesets

Why

Originally only verilog source files were supported. In order to make source code handling more generic, filesets were introduced. Filesets are modeled after IP-XACT filesets and each fileset lists a group of files with similar purpose. Apart from supporting more file types, the filesets contain some additional control over when to use the files. The verilog section is still supported for some time to allow users to perform the migration.

When

`fileset` sections were introduced in FuseSoC 1.4

The `verilog` section is still supported to allow time for migrations

How

Given a `verilog` section with the following contents:

```
[verilog]
src_files = file1.v file2.v
include_files = file3.vh file4.vh
tb_src_files = file5.v file6.v
tb_include_files = file7.vh file8.vh
tb_private_src_files = file9.v file10.v
```

these will be turned into multiple file sets. The names of the file sets are not important, but should reflect the usage of the files.

```
[fileset src_files]
files = file1.v file2.v
file_type = verilogSource

[fileset include_files]
files = file3.vh file4.vh
file_type = verilogSource
is_include_file = true

[fileset tb_src_files]
files = file5.v file6.v
file_type = verilogSource
usage = sim

[fileset tb_include_files]
files = file7.vh file8.vh
file_type = verilogSource
is_include_file = true
usage = sim

[fileset tb_private_src_files]
files = file9.v file10.v
file_type = verilogSource
scope = private
usage = sim
```

If not specified, `usage = sim synth` and `scope = public`

These filesets can be further combined by setting some per-file attributes

```
[fileset src_files]
files =
 file1.v
 file2.v
 file3.vh[is_include_file]
 file4.vh[is_include_file]
file_type = verilogSource

[fileset public_tb_files]
files = file5.v file6.v file7.vh[is_include_file] file8.vh[is_include_file]
file_type = verilogSource
usage = sim

[fileset tb_files]
files = file9.v file10.v
file_type = verilogSource
scope = private
usage = sim
```

`file_type` can also be overridden on a per-file basis (e.g. `file2.v[file_type=verilogSource-2005] file3.vh[is_include_file,file_type=systemVerilogSource]`), but scope and usage are set for each fileset.

2.3.5 Migrating from verilator define_files

Why

Files specified as `define_files` in the verilator core section were treated as verilog files containing `` `define `` statements to C header files with equivalent #define statements. While there are use-cases for this functionality, the actual implementation is limited and makes assumptions that makes it difficult to maintain in the FuseSoC code base. The decision is therefore made to deprecate this functionality and instead require the user to make the conversion.

When

`verilator define_files` are no longer converted in FuseSoC 1.7

How

The following stand-alone Python script will perform the same function. It can also be executed as a `pre_build` script to perform the conversion automatically before a build

```
def convert_V2H( read_file, write_file):
    fV = open (read_file,'r')
    fC = open (write_file,'w')
    fC.write("//File auto-converted the Verilog to C. converted by FuseSoC//\n")
    fC.write("//source file --> " + read_file + "\n")
    for line in fV:
        Sline=line.split('`',1)
        if len(Sline) == 1:
            fC.write(Sline[0])
```

(continues on next page)

(continued from previous page)

```
        else:
            fC.write(Sline[0]+"#"+Sline[1])
    fC.close
    fV.close

import sys
if __name__ == "__main__":
    convert_V2H(sys.argv[1], sys.argv[2])
```

2.3.6 Redefining build_root

Why

As an aid for scripts executed during the build process, a number of environment variables were defined. Unfortunately this was done without too much thought and as time moved on, some of these turned out to be a maintenance burden without bringing much benefit, and in some cases without ever being used.

At the same time, the introduction of VLNV and dependency ranges has introduced non-determinism in where the output of a build ends up. For these reasons, it was determined to redefine the rarely used *build_root* variable to point to the the directory containing the work root and exported files. A *–build-root* command-line switch is introduced to explictly set a build_root. Setting *build_root* in *fusesoc.conf* will keep working the same way as before, but the command-line switch takes precedence. CAPI1 cores will no longer export the *BUILD_ROOT* environment variable.

These changes affects the following cases:

- Relying on the *BUILD_ROOT* variable in scripts called from CAPI1 cores.

When

build_root was redefined after the release of FuseSoC 1.9.1

How

Any scripts that previously relied on `$BUILD_ROOT` will have to be updated. Note that due to other changes in FuseSoC most of them were unlikely to work at this point anyway.

2.4 Glossary

In the context of FuseSoC some terms have special meaning. This glossary section explains some of the jargon used.

core A core is a reasonably self-contained, reusable piece of IP, such as a FIFO implementation. See also *FuseSoC's basic building block: cores*.

core file

core description file A file describing a *core*, including source files, available targets, etc.

semantic versioning

SemVer Semantic versioning is a convention to give meaning to version numbers. See *Semantic versioning (SemVer)* and semver.org.

stage

build stage See *Build stages*.

target See *Targets*.

tool

tool flow See *Tool flows*.

VLNV Vendor, Library, Name, and Version: the format used for *core* names. In core names, the four parts are separated by colons, forming a name like `vendor:library:name:version`.

See also *The core name, version, and description*.

YAML YAML is (among other things) a markup language, commonly used for configuration files. It is used in FuseSoC in various places, especially for *core description files* and for EDAM files.

Read more about YAML on Wikipedia or on yaml.org.

CHAPTER

THREE

FUSESOC DEVELOPER'S GUIDE

3.1 Development Setup

Note: To make changes to a backend, e.g. to the way a simulator or synthesis tool is called, you need to modify edalize, not fusesoc. Edalize is a separate project, see https://github.com/olofk/edalize for more information.

3.1.1 Get the code

The FuseSoC source code is maintained in a git repository hosted on GitHub. To improve FuseSoC itself, or to test the latest unreleased version, it is necessary to clone the git repository first.

```
cd your/preferred/source/directory
git clone https://github.com/olofk/fusesoc
```

3.1.2 Setup development environment

Note: If you have already installed FuseSoC, remove it first using `pip3 uninstall fusesoc`.

To develop FuseSoC and test the changes, the fusesoc package needs to be installed in editable or development mode. In this mode, the `fusesoc` command is linked to the source directory, and changes made to the source code are immediately visible when calling `fusesoc`.

```
# Install all Python packages required to develop fusesoc
pip3 install --user -r dev-requirements.txt

# Install Git pre-commit hooks, e.g. for the code formatter and lint tools
pre-commit install

# Install the fusesoc package in editable mode
pip3 install --user -e .
```

Note: All commands above use Python 3 and install software only for the current user. If, after this installation, the `fusesoc` command cannot be found adjust your `PATH` environment variable to include `~/.local/bin`.

After this installation is completed, you can

- edit files in the source directory and re-run `fusesoc` to immediately see the changes,
- run the unit tests as outlined in the section below, and
- use linter and automated code formatters.

3.1.3 Formatting and linting code

The FuseSoC code comes with tooling to automatically format code to conform to our expectations. These tools are installed and called through a tool called pre-commit. No setup is required: whenever you do a `git commit`, the necessary tools are called and your code is automatically formatted and checked for common mistakes.

To check the whole source code `pre-commit` can be run directly:

```
# check and fix all files
pre-commit run -a
```

3.1.4 Running tests

The FuseSoC contains unit tests written using the pytest framework. To run the tests in an isolated environment it is recommended to run pytest through tox, which first creates a package of the source code, installs it, and then runs the tests. This ensures that packaging and environment errors are less likely to slip through.

```
cd fusesoc/source/directory

# Run all tests in an isolated environment (recommended)
tox

# All arguments passed to tox after -- are passed to pytest directly.
# E.g. run a single test: use filename::method_name, e.g.
tox -- tests/test_capi2.py::test_capi2_get_tool --verbose

# Alternatively, tests can be run directly from the source tree.
# E.g. to run a single test: use filename::method_name, e.g.
python3 -m pytest
```

Refer to the pytest documentation for more information how tests can be run.

Note: In many installations you can replace `python3 -m pytest` with the shorter `pytest` command.

3.1.5 Building the documentation

The FuseSoC documentation (i.e., the thing you're reading right now) is built from files in the `doc` directory in the FuseSoC source repository. The documentation is written reStructuredText, and Sphinx is used to convert the documentation into different output formats, such as HTML or PDF.

Use the following command to build the documentation on your machine after making changes to it. The rendered documentation can be previewed by pointing a browser to the output file as shown in the run output, typically `.tox/docs_out/index.html` in the current directory.

```
cd fusesoc/source/directory
tox -e doc

# On Linux: Open the rendered documentaton with the standard browser
xdg-open .tox/docs_out/index.html
```

INDEX