FuseSoC is a package manager and a set of build tools for HDL code. Its main purpose is to increase reuse of IP cores and be an aid for creating, building and simulating SoC solutions.

1.1 Core description files

capi (Core API) is the format for core description files. Current version is version 2.0. A capi 2.0 file is identified by the string “CAPI=2” in the beginning of a file. The rest of the file is a standard YAML file.

1.2 Core naming rules

FuseSoC uses VLNV tags to uniquely identify a core. VLNV is a concept borrowed from the IP-XACT and stands for Vendor Library Name Version. This means that the name of the cores consists of four parts, which are generally separated by ‘:’, such as librecores.org:peripherals:uart16550:1.5. In FuseSoC, it is allowed to leave out all parts of the VLNV tag except for the name part, e.g. ::uart16550. In those cases, the Vendor and Library parts will be empty strings, and the version will be set to 0.

As the VLNV concept was introduced in FuseSoC after many core files had already been created, FuseSoC still supports parsing files with the legacy naming convention. These can either be of the format name, in which case they will be translated internally to VLNV tags with the Name field set, and Version set to 0, or they can be of the format name-version, which will also set the Version field.

As an extension to the VLNV naming scheme, FuseSoC also support specifying a revision of a core file. This is a fifth field that can be added to both legacy and VLNV names by adding -r<revision> as a suffix (e.g. ::uart16550:1.5-r1, uart16550-1.5-r1, uart16550-r1). This is used to make updates to the .core file even if the source of the core is unchanged.
1.3 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 `%LOCALAPPDATA%\fusesoc` in 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` with the following contents:

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

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

1.4 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 added 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 VLNVIDentifier 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`.

1.5 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

1.6 Backends

FuseSoC uses the backends available from Edalize

1.7 Migration guide

As new features are added to FuseSoC, some older features become obsolete. Read the
link:migrations{outfilesuffix}[migration guide] to learn how to keep the .core files up-to-date with the latest
best practices
2.1 Type definitions

[[File]] 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 <<FileTypes, 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]] FileList ~~~~~~~ Space-separated list of <>

Each element in the list is first subjected to the expansion according to <> and then parsed as a <>

[[PathList]] PathList ~~~~~~~ Space-separated list of paths

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

[[SimulatorList]] SimulatorList ~~~~~~~~~~~ List of supported simulators. Allowed values are ghdl, icarus, isim, modelsim, verilator, xsim

[[SourceType]] SourceType ~~~~~~~~~~~ Language used for Verilator testbenches. Allowed values are C, CPP or systemC

[[StringList]] StringList ~~~~~~~~~~~ Space-separated list of strings

[[VlnvList]] 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 m0r1kx =::i2c:1.14

[[FileTypes]] File types ———-
The following valid file types are defined: PCF, QIP, SDC, UCF, 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 Sections

fileset ~~~~~

 ghdl ~~~~

 icarus ~~~~~

 icestorm ~~~~~

 ise ~~~

 isim ~~~

 main ~~~
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>backend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backend for FPGA implementation</td>
<td>component</td>
<td>Core IP-Xact component file</td>
<td>depend</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;PathList,PathList&gt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Common dependencies</td>
<td>description</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Supported simulators. Valid values are icarus, modelsim, verilator, isim and xsim. Each simulator have a dedicated section described elsewhere in this document</td>
<td>modelsim</td>
<td>~~~~~~~</td>
</tr>
<tr>
<td></td>
<td></td>
<td>parameter</td>
<td>~~~~~~~</td>
</tr>
<tr>
<td></td>
<td></td>
<td>quartus</td>
<td>~~~~~~~</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rivierapro</td>
<td>~~~~~~~</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scripts</td>
<td>~~~~~~~</td>
</tr>
<tr>
<td></td>
<td></td>
<td>verilator</td>
<td>~~~~~~~</td>
</tr>
<tr>
<td></td>
<td></td>
<td>verilog</td>
<td>~~~~~~~</td>
</tr>
</tbody>
</table>
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.

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).
2.3 Known issues

The configparser in python 2 doesn’t handle spaces before values in multiline options. + .Illegal comment style

```
src_files = clkgen.v #gpio.v fusesoc_top.v
```

+ This is not legal in python 2, while:

```
src_files = clkgen.v # gpio.v fusesoc_top.v
```

+ is ok in python 2 and python 3. +

. Spaces are not allowed anywhere in the paths.
CHAPTER 3

CAPI2

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

3.1 Types

3.1.1 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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_include_file</td>
<td>bool</td>
<td>Treats file as an include file when true</td>
</tr>
<tr>
<td>file_type</td>
<td>str</td>
<td>File type. Overrides the file_type set on the containing fileset</td>
</tr>
<tr>
<td>logical_name</td>
<td>str</td>
<td>Logical name, i.e. library for VHDL/SystemVerilog. Overrides the logical_name set on the containing fileset</td>
</tr>
</tbody>
</table>

3.1.2 Genparams

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

3.1.3 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.
3.1.4 String

String 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)]
```

3.1.5 StringOrList

Item is allowed to be either a `String` or a list of `String`.

3.1.6 Vlnv

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

3.2 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

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

3.2.1 Icecube2

Options for Icecube2 backend

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>part</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>package</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>
3.2.2 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.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_tool</td>
<td>String</td>
<td>Default tool to use unless overridden with --tool=</td>
</tr>
<tr>
<td>hooks</td>
<td>Hooks</td>
<td>Script hooks to run when target is used</td>
</tr>
<tr>
<td>tools</td>
<td>Tools</td>
<td>Tool-specific options for target</td>
</tr>
<tr>
<td>toplevel</td>
<td>StringOrList</td>
<td>Top-level module. Normally a single module/entity but can be a list of several items</td>
</tr>
<tr>
<td>filessets</td>
<td>List of String</td>
<td>File sets to use in target</td>
</tr>
<tr>
<td>generate</td>
<td>List of String</td>
<td>Parameterized generators to run for this target</td>
</tr>
<tr>
<td>parameters</td>
<td>List of String</td>
<td>Parameters to use in target. The parameter default value can be set here with param=value</td>
</tr>
<tr>
<td>vpi</td>
<td>List of String</td>
<td>VPI modules to build and include for target</td>
</tr>
</tbody>
</table>

3.2.3 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 always executed from the work root.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>Dict of String</td>
<td>Map of environment variables to set before launching the script</td>
</tr>
<tr>
<td>cmd</td>
<td>List of String</td>
<td>List of command-line arguments</td>
</tr>
<tr>
<td>filessets</td>
<td>List of String</td>
<td>Filesets needed to run the script</td>
</tr>
</tbody>
</table>

3.2.4 Quartus

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>family</td>
<td>String</td>
<td>FPGA family (e.g. Cyclone V)</td>
</tr>
<tr>
<td>device</td>
<td>String</td>
<td>FPGA device (e.g. 5CSXFC6D6F31C8ES)</td>
</tr>
<tr>
<td>quartus_options</td>
<td>List of String</td>
<td>Additional options for Quartus</td>
</tr>
</tbody>
</table>

3.2.5 Modelsim

ModelSim simulator from Mentor Graphics
### 3.2.6 Vivado

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>part</td>
<td>String</td>
<td>FPGA part number (e.g. xc7a35tcsg324-1)</td>
</tr>
</tbody>
</table>

### 3.2.7 Vcs

Options for Vcs backend

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcs_options</td>
<td>List of String</td>
<td></td>
</tr>
</tbody>
</table>

### 3.2.8 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.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>String</td>
<td>The command to run (relative to the core root)</td>
</tr>
<tr>
<td>interpreter</td>
<td>String</td>
<td>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.</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>Short description of the generator, as shown with fusesoc gen list</td>
</tr>
<tr>
<td>usage</td>
<td>String</td>
<td>A longer description of how to use the generator, including which parameters it uses (as shown with fusesoc gen show $generator)</td>
</tr>
</tbody>
</table>

### 3.2.9 Trellis

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextpnr_options</td>
<td>List of String</td>
<td>Additional options for nextpnr</td>
</tr>
<tr>
<td>yosys_synth_options</td>
<td>List of String</td>
<td>Additional options for the synth_ecp5 command</td>
</tr>
</tbody>
</table>
3.2.10 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.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyze_options</td>
<td>List of String</td>
<td>Options to use for the analyze (ghdl -a) phase</td>
</tr>
<tr>
<td>run_options</td>
<td>List of String</td>
<td>Options to use for the run (ghdl -r) phase</td>
</tr>
</tbody>
</table>

3.2.11 Isim

Xilinx ISim simulator from ISE design suite

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fuse_options</td>
<td>List of String</td>
<td>Additional options for compilation with FUSE</td>
</tr>
<tr>
<td>isim_options</td>
<td>List of String</td>
<td>Additional run options for ISim</td>
</tr>
</tbody>
</table>

3.2.12 Parameter

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatype</td>
<td>String</td>
<td>Parameter datatype. Legal values are bool, file, int, str. file is same as str, but prefixed with the current directory that FuseSoC runs from</td>
</tr>
<tr>
<td>default</td>
<td>String</td>
<td>Default value</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>Description of the parameter, as can be seen with fusesoc run --target=$target $core --help</td>
</tr>
<tr>
<td>paramtype</td>
<td>String</td>
<td>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.</td>
</tr>
<tr>
<td>scope</td>
<td>String</td>
<td>Not used : Kept for backwards compatibility</td>
</tr>
</tbody>
</table>

3.2.13 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

```bash
spyglass:
  methodology: "GuideWare/latest/block/rtl_handoff"
  goals:
    - lint/lint rtl
  spyglass_options:
```

(continues on next page)
# 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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>methodology</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>spyglass_options</td>
<td>List of String</td>
<td></td>
</tr>
<tr>
<td>goals</td>
<td>List of String</td>
<td></td>
</tr>
<tr>
<td>rule_parameters</td>
<td>List of String</td>
<td></td>
</tr>
</tbody>
</table>

### 3.2.14 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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timescale</td>
<td>String</td>
<td>Default timescale</td>
</tr>
<tr>
<td>iverilog_options</td>
<td>List of String</td>
<td>Additional options for iverilog</td>
</tr>
</tbody>
</table>

### 3.2.15 Icestorm

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pnr</td>
<td>String</td>
<td>Select Place &amp; Route tool. Legal values are arachne for Arachne-PNR or next for nextpnr. Default is arachne</td>
</tr>
<tr>
<td>arachne_pnr_options</td>
<td>List of String</td>
<td>Additional options for Arachnhe PNR</td>
</tr>
<tr>
<td>nextpnr_options</td>
<td>List of String</td>
<td>Additional options for nextpnr</td>
</tr>
<tr>
<td>yosys_synth_options</td>
<td>List of String</td>
<td>Additional options for the synth_ice40 command</td>
</tr>
</tbody>
</table>

### 3.2.16 Xsim

XSim simulator from the Xilinx Vivado suite

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xelab_options</td>
<td>List of String</td>
<td>Additional options for compilation with xelab</td>
</tr>
<tr>
<td>xsim_options</td>
<td>List of String</td>
<td>Additional run options for XSim</td>
</tr>
</tbody>
</table>

### 3.2.17 Rivierapro

Riviera Pro simulator from Aldec
### 3.2.18 Fileset

A fileset represents a group of files 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.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_type</td>
<td>String</td>
<td>Default file_type for files in fileset</td>
</tr>
<tr>
<td>logical_name</td>
<td>String</td>
<td>Default logical_name (i.e. library) for files in fileset</td>
</tr>
<tr>
<td>files</td>
<td>List of File</td>
<td>Files in fileset</td>
</tr>
<tr>
<td>depend</td>
<td>List of String</td>
<td>Dependencies of fileset</td>
</tr>
</tbody>
</table>

### 3.2.19 Verilator

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>String</td>
<td>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</td>
</tr>
<tr>
<td>cli-parser</td>
<td>String</td>
<td>Select whether FuseSoC should handle command-line arguments (managed) or if they should be passed directly to the verilated model (raw). Default is managed</td>
</tr>
<tr>
<td>libs</td>
<td>List of String</td>
<td>Extra libraries for the verilated model to link against</td>
</tr>
<tr>
<td>verilator_options</td>
<td>List of String</td>
<td>Additional options for verilator</td>
</tr>
</tbody>
</table>

### 3.2.20 Ise

Xilinx ISE Design Suite

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>family</td>
<td>String</td>
<td>FPGA family (e.g. spartan6)</td>
</tr>
<tr>
<td>device</td>
<td>String</td>
<td>FPGA device (e.g. xc6slx45)</td>
</tr>
<tr>
<td>package</td>
<td>String</td>
<td>FPGA package (e.g. csg324)</td>
</tr>
<tr>
<td>speed</td>
<td>String</td>
<td>FPGA speed grade (e.g. -2)</td>
</tr>
</tbody>
</table>

### 3.2.21 Hooks

Hooks are scripts that are run at different points in the build process. They are always launched from the work root.
### 3.2.22 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.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre_build</td>
<td>List of String</td>
<td>Scripts executed before the build phase</td>
</tr>
<tr>
<td>post_build</td>
<td>List of String</td>
<td>Scripts executed after the build phase</td>
</tr>
<tr>
<td>pre_run</td>
<td>List of String</td>
<td>Scripts executed before the run phase</td>
</tr>
<tr>
<td>post_run</td>
<td>List of String</td>
<td>Scripts executed after the run phase</td>
</tr>
</tbody>
</table>

#### 3.2.23 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.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libs</td>
<td>List of String</td>
<td>External libraries to link against</td>
</tr>
<tr>
<td>filesets</td>
<td>List of String</td>
<td>Filesets containing files to use when compiling the VPI library</td>
</tr>
</tbody>
</table>

#### 3.2.24 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.
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generator</td>
<td>String</td>
<td>The generator to use. Note that the generator must be present in the dependencies of the core.</td>
</tr>
<tr>
<td>parameters</td>
<td>Gen-params</td>
<td>Generator-specific parameters. <code>fusesoc gen show $generator</code> might show available parameters</td>
</tr>
<tr>
<td>position</td>
<td>String</td>
<td>Where to insert the generated core. Legal values are <code>first</code>, <code>append</code> or <code>last</code>. <code>append</code> will insert core after the core that called the generator</td>
</tr>
</tbody>
</table>
A series of tutorials for how to use FuseSoC

4.1 FuseSOC: Getting Started

Some time ago I needed to build an emulated GPS device for a project. I reckoned it was easiest to do so with an old FPGA board and since this is turned out to be a pretty straight-forward SoC I also realized it would serve well as a tutorial. So here it is, the first tutorial on getting started with building systems using FuseSoC.

To help with the project I got my trusty old DE0 Nano board, but most of the tutorial requires no hardware at all, and some parts will work on other hardware with only minor modifications.

4.1.1 Background

The GPS I intend to emulate has two outputs. One is a signal that changes polarity once a second emulating a 0.5Hz clock signal. This is called a PPS (Pulse Per Second) signal in GPS parlour. There are multiple variants of the PPS scheme, in some cases it is a short pulse every second instead of just changing polarity, but we ignore those for now.

The second signal is a serial UART data signal that transmits an ASCII message at 4800 baud rate every second. The data sent on the serial link uses the NMEA protocol. While many messages are defined in the NMEA protocol, this particular device emulator will only implement the $GPGGA message. Each message is sent some time (e.g. 200ms) after the positive edge on the PPS signal.

Both signals together will look something like this

Fig. 1: gps signal graph

Alright, shouldn’t be too hard. Let’s get started with the coding.
4.1.2 Preparations

Before we do anything, we need to install the required tools. First on the list is of course FuseSoC. This is pretty easy using the python pip package manager.

```
pip install fusesoc
```

FuseSoC is now installed, but as with all package managers, they are not all that useful without actual packages. This will become evident if you run

```
fusesoc list-cores
```

This should result in an empty list and a message saying `ERROR: cores_root is not defined` if you never had FuseSoC on your system before.

Luckily FuseSoC comes with two standard libraries (It should really only be one, but the transition has taken a bit longer than expected).

To download and register the standard libraries with FuseSoC run

```
fusesoc init
```

Running `fusesoc list-cores` again should now result in a much larger list of cores.

During `fusesoc init`, FuseSoC has created a configuration file in `$XDG_CONFIG_HOME/fusesoc/fusesoc.conf` ($XDG_CONFIG_HOME is normally set to `~/.config` on Linux systems and somewhere else, not sure to be honest, on Windows and MacOS)

The file will have two sections, one for each of the above mentioned libraries and looks like this

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

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

Sections that start with `library. ` describe a library. There are other types of sections (well, at least one more type) as well, but we will ignore that for now.

What we can see from the configuration file is that each library has a sync-uri and a sync-type option. FuseSoC uses these when a user runs `fusesoc update` to know how and where to get the latest version of the library.

The libraries themselves are found under `$XDG_DATA_HOME/fusesoc` ($XDG_DATA_HOME is normally set to `~/.local/share` under Linux and somewhere else on other operating systems).

We never told FuseSoC explicitly where to find the libraries. The magic part here is that FuseSoC will look at each `[library.<name>]` section and search for a corresponding library under `$XDG_DATA_HOME/fusesoc/<name>`. An explicit location can also be set for a library using the location option. We will do that soon, but let us first look at the contents of the libraries.

The `fusesoc-cores` library will contain a bunch of directories, each containing files ending with `.core`. These files are the heart and soul of FuseSoC and are called core description files. A core description file contains all the information FuseSoC needs to have about the core in order to run simulations on the core, build it for an FPGA target or use it as a dependency of another core. To get familiar with `.core` files, we can take a look at a simple one. Let’s start with `fusesoc-cores/uart16550/uart16550-1.5.5-r1.core`
The information in the core description file is stored in the yaml format with an additional requirement that it must start with `CAPI=2`. The `:` at the end of the first line is a compromise. It’s not used for anything more than making this a valid yaml file.

Next line is the name. Names are specified in the VLNV (Vendor Library Name Version) format with `:` to separate the fields. FuseSoC allows vendor and library to be left empty, which is why many core names start with `::`. The VLNV format comes from IP-XACT, a standard that we hopefully will revisit in later tutorials. For now we can happily ignore that.

Description should be pretty self-explanatory.

Next up is the filesets section. Related source files are lumped together in filesets. There can for example be one fileset for the testbench and another one for the RTL implementation. This example only has a single fileset that is called `rtl`. The file `uart_defines.v` is noted to be an include file, i.e. a file that is included in other verilog files with the `include` statement. All files in the fileset are of the type `verilogSource`.

Moving down we find a section called targets. Targets in core files are a bit like `Makefile` targets, and all settings specified in a specific target section will be used when that target is invoked. Only a single target, namely the default target, is defined here. default is a special target that is used when no explicit target is requested, and this is also the target that will be referenced when this core is used as a dependency of another core. More about dependency handling later. The only thing we do in the default target is to say that this target uses the rtl fileset that was defined above.

Ok, so the core description file references a bunch of files, but... there are no files to be seen anywhere. What’s going on here? To answer that we need to look at the next section, the provider section. If a core description file has a provider section, it’s called a remote core. If it hasn’t, then we call it a local core. When a remote core is needed, FuseSoC will first look in its cache directory to see if it has already been downloaded (fetched). If not, it will look at the provider section to figure out how to fetch the source code. Once it is in the cache, it will use the cached version.

Before we start writing our own first core we will look at a slightly more complicated example.
Let’s take a look at `fusesoc-cores/i2c/i2c-1.14-r1.core` which looks like this

```plaintext
CAPI=2:
name : ::i2c:1.14-r1
filesets:
  rtl_files:
    files:
    - rtl/verilog/i2c_master_bit_ctrl.v
    - rtl/verilog/i2c_master_byte_ctrl.v
    - rtl/verilog/i2c_master_defines.v: {is_include_file : true}
    - rtl/verilog/i2c_master_top.v
    file_type : verilogSource
  tb_files:
    depend:
    - ">=vlog_tb_utils-1.0"
    - wiredelay
    files:
    - bench/verilog/wb_master_model.v
    - bench/verilog/tst_bench_top.v
    - bench/verilog/i2c_slave_model.v
    file_type : verilogSource

targets:
  default:
    filesets : [rtl_files]
sim:
  default_tool : icarus
    filesets : [rtl_files, tb_files]
toplevel : tst_bench_top

provider:
  name : github
  user : olofk
  repo : i2c
  version : v1.14
  patches : [files/0001-add_vlog_tb_utils.patch]
```

Again, we see the `CAPI=2` header, name, filesets, target and the provider section. Let’s focus on the differences from the previous example. In the filesets section, there are now two filesets. The one called `tb_files` also has an additional field called depend. This is where we enter the package management territory of FuseSoC. This means that the files in the `tb_files` fileset uses functionality from other cores and need to have them present when building the project. In this case it requires any version of a core named wiredelay, and at least version 1.0 of a core named `vlog_tb_utils`. Running `fusesoc list-cores` will hopefully reveal that both these cores are present in the standard libraries.

In the target section there is also a new target called sim. This one has two options in addition to the previously mentioned filesets. `default_tool` decides which tool to use if the user doesn’t explicitly selects a tool on the FuseSoC command-line. More on that later. The other option is toplevel. This identifies which verilog module or VHDL entity that should be used as the top-level instance when building the project. Commonly there is a single toplevel, but in some cases several toplevels must be set, in which case this will be defined as a list, e.g. `[first_toplevel, second_toplevel]`.

A third difference is that patches option in the provider section. Sometimes when packaging third-party cores there might be aspects of the original code that does not work well with FuseSoC. By specifying diff files in the patches option, it is possible to apply patches to cores after they have been downloaded before they are stored in the FuseSoC cache. Common uses for this is to remove hard-coded file paths that clashes with FuseSoC directory layout or add useful features. In this case the patch adds support for functionality from the `vlog_tb_utils` core.
We are now almost ready to write our own core, but let’s begin with running some examples related to the core we just looked at. Start by running `fusesoc core-info`. There’s not much information here, but we should be able to verify that it’s indeed the core we looked at by checking the line starting with `Core root`. We can also see that the FuseSoC core parser has found two targets, default and sim.

Next up we can run the testbench of the `i2c` core. Before doing this, create an empty directory that will be used as the workspace. For this tutorial we will use `~/fusesoc_tutorial/workspace`. Enter the newly created workspace directory, make sure Icarus Verilog is installed and run `fusesoc run --target=sim i2c`. This should run the testbench and finally output `Testbench done`.

If we look at the beginning of the output from the command we will see

```
INFO: Preparing ::vlog_tb_utils:1.1
INFO: Downloading fusesoc/vlog_tb_utils from github
INFO: Preparing ::wiredelay:0
INFO: Preparing ::i2c:1.14-r1
INFO: Downloading olofk/i2c from github
```

When we launched the simulation, we specified `--target=sim`, which uses the `targets/sim` section in the core file. Going deeper down the rabbit hole, the sim section uses the `tb_files` fileset and the `tb_files` fileset depend on `>=vlog_tb_utils-1.0` and `wiredelay`. The `>=vlog_tb_utils-1.0` requirement got us `vlog_tb_utils:1.1`, which is the highest version to satisfy this requirement. This is also a remote core, which means that FuseSoC had to download it and put it in the cache before it could be used. `wiredelay` on the other hand is a local core and all the needed files are already present on the disk (Check `core-info wiredelay` to find out where the files are located). Also the `i2c` core was a remote core and had to be downloaded. You will now find the cached cores in `$XDG_CACHE_HOME/fusesoc` (normally `~/.cache/fusesoc` on Linux systems).

If another simulator is preferred, we can use that by adding a `--tool=` option on the command line, e.g. `fusesoc run --target=sim --tool=modelsim i2c`.

Looking at our workspace directory there will now be a directory called `build` containing a directory called `i2c_1.14-r1`. Inside of that you will find sim-icarus and src. src contains all the sources of the cores in the dependency tree (i.e. `vlog_tb_utils`, `wiredelay` and `i2c`). sim-icarus is the working directory of the simulation, meaning the directory where the simulator tool was launched from. There are a couple of files in there, and these will be revisited in later tutorials. The only thing to notice right now is that the directory itself is named after `<target>-<tool>`.

Other simulators to try are isim, xsim or rivierapro. Be aware that not all tools work for all cores. Upon running a second simulation there should also not be any `INFO: Downloading...` messages as the dependencies are already cached.

Let’s run another simulation, but this time we want to generated a VCD waveform file. Run `fusesoc run --target=sim i2c --vcd`. Looking at the work directory now, there will also be a file called `testlog.vcd` which can be viewed with GTKWave or other VCD-compatible readers. One important thing to be aware of here is that the options specified after the core we want to run (i2c in this case) are options that are specified by the cores themselves. FuseSoC has no knowledge of a `--vcd` option but will just pass it along. In this case it’s not even the i2c core that specifies the `--vcd` option, but the `vlog_tb_utils` core that we depend on. How this works will be explained later, but to know what options that is implemented for each tool, target and core combination run `fusesoc run <core> --help`.

Running `fusesoc run --help` instead will reveal the options available for the run command and running `fusesoc --help` will show available commands and global options.

### 4.2 FuseSoC: Creating a Core

We covered the basics of core files and FuseSoC usage in tutorial 1. It’s now time to use this knowledge to create our own core.
We will start by creating the PPS functionality described in tutorial 1. To get some indication if this works as intended we can connect the output to a LED (which should hopefully exist on every development board ever made). I will call this never-seen-before technological marvel Blinky (patent pending for “Device for changing luminosity of a solid state light source at a fixed interval”).

First thing to do is to create a new directory and put a file inside with a .core. We will use the directory ~/fusesoc_tutorial/cores/gpsemulator and name the file blinky.core.

blinky.core will look like this to begin with:

```
CAPI=2:
name : ::blinky:0
```

This is the minimal information we need in a core file. The CAPI header and a name for the core. If you now try to run fusesoc list-cores or fusesoc core-info blinky you will notice that the core cannot be found. This is because we haven’t yet told FuseSoC where to look for the core.

There are two and a half way to make FuseSoC find the core. The first method is to specify it on the command-line. Assuming the aforementioned directory layout and that we run from the previously created workspace directory we can run fusesoc --cores-root=../cores core-info blinky. As FuseSoC will recursively search for cores in a given path, it’s fine to use the parent directory of the one containing the .core file. In fact, as we will grow our collection of cores over time it can be a good idea to put them all under the same parent directory to avoid having to specify many paths.

The –cores-root argument is transient, and FuseSoC will have forgotten all about it after it has finished its work. To avoid having to specify that argument all the time, we will instead register the directory as a library in a configuration file. One or more cores in a directory tree is called a library in FuseSoC terminology. To register the directory run

```
fusesoc library add tutorial ../cores
```

Now you should be able to run fusesoc core-info blinky and FuseSoC will find the core.

Running fusesoc library add will have created a file called fusesoc.conf in the workspace directory looking something like this:

```
[library.tutorial]
sync-type = local
location = /home/user/fusesoc_tutorial/cores
```

We will not do an in-depth analysis of the configuration file right now, but just notice that every library has a name, passed as the first argument to fusesoc library add and in this case also a location. Now, if you run fusesoc list-cores there will be a lot more cores than just the one we added. Also, if you run fusesoc list-cores from another directory than the workspace directory, it will not find good ol’ blinky. This is because FuseSoC normally reads the libraries from $XDG_CONFIG_HOME/fusesoc/fusesoc.conf and then from fusesoc.conf in the current directory. This makes it easy to have the cores from the standard libraries always available, but use different project-specific cores on top of that by using different workspaces. Running fusesoc --config=<config file> is a special case where only the libraries specified in that configuration file is used. If we run fusesoc --config=fusesoc.conf list-cores it will only list our blinky core. Finally, to add a library to the global configuration file, just run fusesoc library add --global <name> <location>. That’s enough about configuration files for a while. Let’s start coding.

Now we can create the verilog module in pps.v within the gpsemulator core directory. Blinky needs a clock input, an output and an indication of the clock frequency to know how fast it should blink. It should look somewhat like this, assuming a default clock frequency of 50MHz
Before testing this in hardware we should run a simulation. Therefore we need a testbench. Testbenches usually follow the convention of `modulename_tb.v`. In our case we will add the following code to `pps_tb.v` within the gpsemulator core directly, next to the `pps.v` that we created before. This will serve as our initial testbench. We will gradually improve the design of the testbench to showcase some FuseSoC features.

```
`timescale 1ns/1ns
module pps_tb;

  parameter clk_freq_hz = 50_000;
  localparam clk_half_period = 1000_000_000/clk_freq_hz/2;

  reg clk = 1'b1;
  always #clk_half_period clk <= !clk;

  wire pps;
  pps
    #(.clk_freq_hz (clk_freq_hz))
    dut
      (.clk  (clk),
       .pps_o (pps));

  integer i;
  time last_edge = 0;

  initial begin
    @(pps);
    last_edge = $time;
    for (i=0; i<10;i=i+1) begin
      @(pps);
      if ((time-last_edge) != 1_000_000_000) begin
        $display("Error! Length of pulse was %0d ns", $time-last_edge);
        $finish;
      end else
        $display("Pulse %0d/10 OK!", i+1);
      last_edge = $time;
    end
    $display("Testbench finished OK");
    $finish;
  end
endmodule
```
The code will create a 50MHz clock and check that the pps pulse changes every second (1,000,000,000 ns) for ten pulses. Also note that as the testbench will take forever to execute with a simulated 50MHz clock, we have changed the default clock frequency to 50KHz for now. Save it as pps_tb.v in the core directory.

In order to let FuseSoC run a simulation, we need to add the files to our .core file. As pps.v and pps_tb.v have different purposes, with one being the RTL code and one being a testbench we will put them in different filesets. At this point it’s not really necessary, but it will become apparent later why we choose to do this. Also note that the names of the filesets have no meaning. Just choose names that describe the intent. Add the following snippet to the .core file to create two new filesets.

```
filesets:
  rtl:
    files:
      - pps.v : {file_type : verilogSource}
  tb:
    files:
      - pps_tb.v : {file_type : verilogSource}
```

Next, we need to create a target that uses the filesets and also tell the simulation tool which module is our toplevel. We will use sim (for simulation) as the name of the target. Target names can also be chosen mostly arbitrarily, but there are some rules that we will come back to later. This snippet will set up our new target

```
targets:
  sim:
    filesets : [rtl, tb]
    toplevel: [pps_tb]
```

With the complete core file looking like

```
CAPI=2:
  name : ::blinky:0
filesets:
  rtl:
    files:
      - pps.v : {file_type : verilogSource}
  tb:
    files:
      - pps_tb.v : {file_type : verilogSource}
targets:
  sim:
    filesets : [rtl, tb]
    toplevel: [pps_tb]
```

we are ready to go. Run fusesoc run --target=sim --tool=icarus blinky to build the simulation model and run it with Icarus Verilog.

In the next tutorial we will start experimenting with the options at our disposal to learn about some more FuseSoC features.
4.3 FuseSoC: Tool options

In the last tutorial we created a simple core with a testbench and executed this. It’s now time to look at some features that make it more obvious why we want to use FuseSoC instead of just writing a script that launches Icarus Verilog with our two verilog files. Perhaps the most immediately useful thing is to take advantage of FuseSoC’s role as an abstraction layer for different EDA tools. In tutorial 1 we saw how to run a testbench in different simulators by just changing the –tool option to modelsim, xsim, isim or rivierapro. That won’t work this time however. Running with modelsim or xsim will produce errors from the tools complaining that timescale is not set for all units. Verilog’s system of time units and precision is quite annoying and highly tool-specific. We won’t dig deeper into that now, but just present a way to get around the problem by specifying a default timescale for the complaining tools. Now, FuseSoC doesn’t have a dedicated way to set timescale, but any tool-specific options can be supplied through a special tools section. To set a default timescale for modelsim and xsim we would add this to our .core file targets: sim: tools: modelsim: vlog_options: [-timescale=1ns/1ns] xsim: xelab_options: [-timescale, 1ns/1ns]

Running the simulation with modelsim or xsim should now work. Other common uses of tool-specific options is to suppress certain warnings or include vendor-provided libraries. We now know that the testbench works with multiple tools, but we most of the time we are happy to use Icarus Verilog. To save us from writing --tool=icarus all the time we will therefore set a default tool for our sim target and the complete core file will at this point look like this:

```
CAPI=2:
name : ::blinky:0

filesets:
  rtl:
    files:
    - pps.v : {file_type : verilogSource}

  tb:
    files:
    - pps_tb.v : {file_type : verilogSource}

targets:
  sim:
    default_tool: icarus
    filesets : [rtl, tb]
    tools:
      modelsim:
        vlog_options: [-timescale=1ns/1ns]
      xsim:
        xelab_options: [-timescale, 1ns/1ns]
    toplevel: pps_tb
```
to look like this:

```yaml
tb: files: - pps_tb.v : {file_type : verilogSource} depend: [vlog_tb_utils]
```

The `depend` section is a list of VLNV entries we want to depend on. In the most basic form it will depend on the latest version of `vlog_tb_utils`. Running `fusesoc list-cores` will show all the available versions of each core. If we want to use a specific version of a core, we can add a version modifier, such as “>=vlog_tb_utils-1.1”, “<vlog_tb_utils-1.1” or “=vlog_tb_utils-1.0”. Specifying an exact version of a core is generally discouraged if other versions work too, as there will be conflicts when resolving dependencies if one core depends on `=vlog_tb_utils-1.0` while another depends on `=vlog_tb_utils-1.1`. If those are the minimum versions required, it’s instead better to depend on `>=vlog_tb_utils-1.0` and `>=vlog_tb_utils-1.1` since it will allow FuseSoC to find a version (1.1) that works for both cases. Some other rules regarding version and name matching to be aware of:

- `core_name` is a short-hand notation for the VLNV identifier `::core_name`
- `core_name` will depend on the latest version of the core
- `core_name-version` will depend on that exact version. If using VLNV names, the syntax is instead `::core_name:version`
- The version modifiers are `<,<=,=,>,>=`
- Due to yaml parsing, using version modifiers require the whole string to be quoted, e.g. “>=core_name-version”

We can now run `fusesoc run --target=sim --tool=icarus blinky --help` to see some new options available.

Just adding the core as a dependency doesn’t do much difference though. We also need to make use of it. `vlog_tb_utils` exposes a module called `vlog_tb_utils` that we will now add to the testbench.

Add the following line to the testbench to instantiate the helper module.

```verilog
vlog_tb_utils vtu();
```

Running `fusesoc run --target=sim blinky --vcd` will now create a file called `estlog.vcd` in `build/blinky_0/sim-icarus`.

Feel free to try all the target-specific options with different tools, but beware that some of the options won’t do anything at this point and other will behave very differently between tools. We will return to that later, but our next task will be to introduce parameters before we can finally see our code running on real hardware. Until then, our core file should look like this

```
CAPI=2:
name : ::blinky:0

filesets:
  rtl:
    files:
      - pps.v : {file_type : verilogSource}
  
  tb:
    files:
      - pps_tb.v : {file_type : verilogSource}
    depend: [vlog_tb_utils]

targets:
  sim:
    default_tool: icarus
    filesets : [rtl, tb]
    tools:
      modelsim:
```

(continues on next page)
vlog_options: [-timescale=1ns/1ns]
xsim:
  xelab_options: [--timescale, 1ns/1ns]
toplevel: pps_tb
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.

5.1 Migrating from .system files

5.1.1 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.

5.1.2 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

5.1.3 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.

5.2 Migrating from plusargs

5.2.1 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.

5.2.2 When

Parameter sections were introduced in FuseSoC 1.3

The plusargs section is still supported to allow time for migrations

5.2.3 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).

5.3 Migrating to filesets

5.3.1 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.

5.3.2 When

Fileset sections were introduced in FuseSoC 1.4

The verilog section is still supported to allow time for migrations
5.3.3 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.

```yaml
[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 file sets can be further combined by setting some per-file attributes

```yaml
[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

[ifset tb_files]
```
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.

5.4 Migrating from verilator define_files

5.4.1 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.

5.4.2 When

verilator define_files are no longer converted in FuseSoC 1.7

5.4.3 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

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

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

- genindex
- modindex
- search