ZlibValidation

Generated on Tue Apr 15 2025 20:55:05 for ZlibValidation by Doxygen 1.9.5

Tue Apr 15 2025 20:55:05

Chapter 1

ZlibValidation

1.1 Description

Command line tool to validate standard cell libraries in `.lib` format.

`ZlibValidation` provides a suite of tools for engineers working with digital IC standard cell libraries in the Liberty format (.lib). It helps ensure library quality, consistency, and facilitates comparison and conversion tasks. Built with C++ for performance and leveraging robust libraries for parsing and command-line interaction.

1.2 Table of Contents

1.3 Motivation

Validating standard cell libraries (.lib) is critical in chip design, but current solutions pose challenges. Commercial tools are expensive, locking out many users (students, startups, researchers), and their closed-source nature prevents customization and hinders innovation. This creates a gap, especially for those needing rapid validation within tight resource constraints.

`ZlibValidation` aims to fill this gap. It is a **free, open-source command-line tool** designed for comprehensive and efficient .lib file validation. Our goals are to:

- Lower the barrier to entry for library quality assurance.
- Provide **fast**, **reliable checks** (parsing, consistency, function, parameters, comparison).
- Enable **transparency**, **customization**, and **community collaboration** through open source.
- Bridge the divide between academic research and industrial needs in validation technology.

1.4 Installation

`ZlibValidation` is designed to be easily built and run directly from source **without requiring administrator** (`sudo`) **privileges** on Linux systems. This makes it ideal for environments where users have restricted permissions. The recommended installation method is building from source.

1.4.1 Prerequisites

Before you begin, ensure you have the following installed on your **Linux** system:

- **Git:** To clone the repository.
- **C++ Compiler:** C++20 standard support required (e.g., GCC >= 12). Developed with GCC 14.2.0.
- **CMake:** Version 3.25 or higher (versions after 4.0.0 may exhibit compatibility issues).
- **Ninja** (recommended) or **Make**: Build systems. Ninja is preferred for its speed.

1.4.2 Optional Pre-requisites

- **ccache:** Optional but recommended for speeding up recompilation. Install it via your package manager (e.g., `conda install conda-forge::ccache`).
- **lld:** Optional but recommended for faster linking. Install it via your package manager (e.g., `conda install conda-forge::lld`).

1.4.3 Building from Source (Recommended Method)

Follow these steps to compile ZlibValidation in your user directory:

1. **Clone the repository:**
   ```
   git clone https://github.com/Cedar17/ZlibValidation.git
   cd ZlibValidation
   ```
2. **Configure the build using CMake:** Create a build directory and run CMake from within it. This keeps build files separate from your source code.
   ```
   mkdir build
   cd build
   # Use Ninja as the build system (recommended):
   cmake .. -G Ninja
   # or if you prefer Make:
   cmake .. -G "Unix Makefiles"
   ```
   Note: No sudo is needed here.
3. **Compile the project:** Use ninja to build the executable. If you chose Make, use make instead, the -j flag speeds up compilation by using multiple processor cores.
   ```
   # If you used Ninja:
   ninja
   # or if you used Make:
   make -j$(nproc) # Adjust $(nproc) or use a specific number like -j8 for multi-threading
   ```
4. **Locate the Executable:** Upon successful compilation, the zlibvalidation executable will be located inside the build directory: ./build/zlibvalidation.
   ```
   # Check the executable version:
   ./zlibvalidation --version
   ```

1.4.4 Running ZlibValidation

Since we haven't performed a system-wide install (which would typically require sudo), you need to run the executable using its path relative to your current location.

- If you are in the project's root directory (ZlibValidation/), run it like this:
  ```
  ./build/zlibvalidation --help
  ```
- If you are inside the build directory, run it like this:
  ```
  ./zlibvalidation --help
  ```

1.4.5 (Optional) Adding to your PATH

For convenience, you can temporarily add the build directory to your shell's PATH variable for the current session:

```
export PATH="$PWD/build:$PATH" # Assumes you are in the ZlibValidation root directory
# Now you can run it directly:
zlibvalidation -h
```

To make this change permanent, you can add the above line to your shell's configuration file (e.g., ∼/.bashrc, ∼/.zshrc).

1.5 Help Message / Features

```
ZlibValidation
Usage: ./zlibvalidation [OPTIONS] [SUBCOMMAND]
Options:
  -h,--help                   Print this help message and exit
  -v,--version                Display program version information and exit
Subcommands:
  parse                       Parse the Liberty file and write JSON to a file
  mono                        Check the monotonicity of timing arc values
  compare                     Compare the comparison library against the reference one and report differences
  supercell                   Generate supercells for the given Liberty file
  zlibboost                   ZlibBoost - Multi-threaded Library Processing Tool
  clear                       Clear the log, JSON, map, markdown, Verilog, SPICE files in this directory
  verilog                     Generate Verilog file for given Liberty file
  spice                       Generate SPICE file for given Liberty file
  func                        Check functional equivalence of two Liberty files or Verilog files
```

1.6 Example Usage

See test directory. There are some shell scripts showing how to use `ZlibValidation`. You can modify them to validate your own standard cell library pdk.

1.7 Documentation and Reference Manual

See https://cedar17.github.io/ZlibValidation/ for HTML documentation and PDF reference manual.

1.7.1 Doumentation Generation

To generate the documentation, you need to have Doxygen and Graphviz installed. You can install them via your package manager or using conda:

```
conda install -c conda-forge doxygen graphviz
```

Then, run the following command in the root directory of the project:

```
doxygen Doxyfile
```

This will generate the documentation in the `doc_doxygen` directory. You can open the `doc_doxygen/html/index.html` file in your web browser to view the documentation.

Alternatively, you can generate the PDF reference manual by LaTeX.

```
cd doc_doxygen/latex
make
```

1.8 Acknowledgements

`ZlibValidation` leverages the power of several excellent third-party libraries, build tools, and external utilities. We extend our sincere gratitude to the developers and communities behind these projects, which were instrumental in building this tool:

1.8.1 Core Functionality Libraries:

- Liberty Parser (csguth/LibertyParser):
  - **Purpose:** Provides the core functionality for parsing the `.lib` (Liberty) standard cell library format. This open-source implementation forms the basis of our library reading capabilities.
  - **Integration:** The source code was cloned from the repository, manually compiled into a static library (`libsi2dr_liberty.a`), and included directly within this project's `include_3rd_party` directory.
  - **License:** SYNOPSYS Open Source License Version 1.0.
- CLI11:
  - **Purpose:** Enables the robust, feature-rich, and user-friendly command-line interface, handling subcommands and options parsing.
  - **Integration:** Managed via CMake's `FetchContent` mechanism during the build process.
  - **License:** BSD 3-Clause License.
- spdlog:
  - **Purpose:** Provides a highly efficient and flexible library for structured logging throughout the application, critical for debugging and user feedback.
  - **Integration:** Managed via CMake's `FetchContent`.
  - **License:** MIT License.
- nlohmann/json:
  - **Purpose:** Used extensively for representing the parsed Liberty library data internally as JSON objects, facilitating data storage, retrieval, and manipulation (e.g., for monotonicity checks and comparisons).
  - **Integration:** Managed via CMake's `FetchContent`.
  - **License:** MIT License.
- ZlibBoost:
  - **Purpose:** A multi-threaded library processing tool which is the foundation of the `zlibboost` subcommand. It significantly boost standard cell library characterization with machine learning using Python. See https://doi.org/10.1145/3658617.3703638 for detailed paper.
  - **Integration:** Need to specify the path to the `ZlibBoost` main python script and python environment in the `ZlibValidation` command line.
  - **License:** BSD 3-Clause License with Commercial Use Restriction.
- tabulate:
  - **Purpose:** Generates formatted text-based tables, significantly improving the readability of reports generated by the `compare` subcommand.

  - **Integration:** Managed via CMake's `FetchContent`.
  - **License:** MIT License.

- `slang`:
  - **Purpose:** A powerful library for parsing SystemVerilog/Verilog code. It's used here to analyze Verilog netlists, extract Abstract Syntax Trees (AST), generate structural Verilog (`verilog` subcommand), and extract logic for functional equivalence checks (`func` subcommand).
  - **Integration:** Managed via CMake's `FetchContent`.
  - **License:** MIT License.

- `ExprTk (Expression Toolkit Library)`:
  - **Purpose:** A fast mathematical expression parser and evaluation engine. It is used in the `func` subcommand to dynamically evaluate the logical function strings extracted from libraries or Verilog, enabling the functional equivalence check by comparing truth tables.
  - **Integration:** The header file (`exprtk.hpp`) is included directly within this project's `include_3rd_party` directory.
  - **License:** MIT License.

1.8.2 Build, Documentation, and External Tools:

- `CMake`: The cross-platform build system generator used to configure and manage the entire compilation process.
- `Doxygen` & `Graphviz`: Employed for automatically generating source code documentation and visualizing relationships, aiding development and understanding.
- `V2LVS (Calibre® Utility)`:
  - **Purpose:** A command-line tool included with the Siemens EDA Calibre® platform, designed to translate structural Verilog netlists into a basic SPICE format, typically for Layout Versus Schematic (LVS) verification.
  - **Integration:** The `spice` subcommand **optionally** utilizes `v2lvs` if it's found in the system's PATH. It serves as the first-pass engine to convert the intermediate Verilog representation into a raw SPICE netlist. `ZlibValidation` then performs post-processing on this output, and ensure correct formatting.

The seamless integration facilitated by CMake FetchContent for many of these libraries, combined with the direct inclusion of others, significantly streamlined development and enabled the rich feature set of `ZlibValidation`. We encourage users to consult the individual project licenses for detailed terms of use.

Chapter 2

Development Diary

2.1 2025-01

2.1.1 2025-01-27

- 创建了一个C++空项目，使用CMake管理编译链，并手动添加了外部库 `libsi2dr_liberty.a`。
- 添加了 `CLI11` 库以处理命令行参数解析。
- 添加了 `spdlog` 日志库，实现高效的日志格式化输出。

2.1.2 2025-01-28

- 添加 `nlohmann/json` 库用于 JSON 数据存储和检索。
- 在 `version.h` 中存储项目作者、版本、构建日期等信息，由 `CMake` 自动维护。
- 简化了 `--version` 参数解析的代码实现，并添加了 `--mode` 选项以选择工作模式。
- 使用 `spdlog` 库的日志记录器将 debug 级别及以上的日志输出到文件，将 info 级别及以上的日志输出到控制台。
- 将库文件顶层封装为 `LibFile` 对象，提供了读取、解析和修改库文件的方法，包含名称和 PVT 信息作为公共属性。
  - 但仍然使用过程化方法循环迭代读取库文件名称和 PVT 信息。
- 按照 C++ 标准库、第三方库和本地库的顺序重新排列了 `#include` 语句。

2.2 2025-02

2.2.1 2025-02-01

- 使用面向对象编程（OOP）重构了原先用C 语言循环遍历的代码，将简单属性的读取封装为 LibAttribute 类，
  - 并将函数返回的 char * 类型的属性值转换为 std::string 类型，便于外部使用。
- 将属性迭代过程封装为 AttributesIterator 类，在析构函数中处理迭代退出，实现资源获取即初始化（RAII）。

2.2.2 2025-02-10

- 将组的迭代过程封装为 GroupsIterator 类，同样在析构函数中处理迭代退出。
- 封装了 LibGroup 类，提供了读取组名、类别、属性、子组的方法。
- 将类的定义和实现分离，将类的实现放在 src 目录下，将类的定义放在 include 目录下，修改了 CMakeLists.txt 自动收集文件，便于模块化编译。

2.2.3 2025-02-11

- 类的头文件改为 hpp 后缀，修改了 CMakeLists.txt 中的文件收集规则，CMake 编译时间戳更新。
- 为 LibFile 类添加了输出同名 json 格式文件的方法 LibFile::writeJsonToFile，目前能输出 PVT、cell_name、cell_footprint、cell_area 信息。
- [x] voltage 浮点数误差未解决。

2.2.4 2025-02-14

- 在 json_utils.cpp 中，实现了以 JSON 数据结构循环迭代存储库 (lib) 信息的功能，具体包括 generateCellJson，generatePinJson，generatePowerJson，generateLutJson 等函数，并添加了同名的 hpp 头文件。
- [x] 时序 (timing) 信息解析功能待完成。
- 新增辅助函数 parseStringToVector，该函数可将以逗号分隔的字符串解析为浮点数向量，以便于读取 Look Up Table (LUT)。
- 为 LibFile 对象添加了私有属性 lib_json_，用于存储库 (lib) 的 JSON 对象。
- 封装了用于迭代复杂属性值的 ValuesIterator 类。

- 修改了 LibAttribute::isComplex() 函数的返回值类型，由原类型变更为 bool 类型。
- 重写了 LibGroup::getName() 函数，使其直接返回 std::string 类型的组名字。
- 重构了迭代器使用代码。移除了中间变量类似 si2drGroupsIdT sub_groups 的声明步骤。直接使用 lib_group.getGroups() 的返回值进行初始化 GroupsIterator，提升了代码的可读性和简洁性。

2.2.5 2025-02-15

- 实现 generateTimingJson 函数，用于全面解析时序信息。

2.2.6 2025-02-17

- 简化了迭代器在外部的调用流程，删除所有迭代器的 begin() 方法，改为在构造函数内部初始化。

2.2.7 2025-02-18

- 新建 LibFile::mono() 方法，准备实现库文件输出引脚的 values 单调递增性检查。
- 修改了 setupLogger() 函数，将日志文件名可作为参数传入，便于根据不同工作模式切换日志文件。

2.2.8 2025-02-19

- 修改了 mode == "mono"情况下的逻辑，使用 C++17 引入的 std::filesystem::exists() 函数检查同名 JSON 文件是否存在，如果 JSON 文件不存在，则先调用库文件解析功能，生成 JSON 文件，然后再执行单调性检查。
- 实现了 LibFile::mono() 方法，检查 cell_rise、cell_fall、rise_transition、fall_transition 这四种 timing 信息，针对其 LUT 数据结构，检查其 value 值在表格的每一行是否保持单调递增。

2.2.9 2025-02-20

- LibFile::mono() 方法增加结果统计功能，在程序运行的最后输出单调性检查中 passed (通过) 和 failed (失败) 的 cell 数量，输出格式参考 liberate_lv 工具的风格。
- 以 sl018_ff_3.96_-40.lib 为例，与 liberate_lv 工具对比测试，结果均为 205 out of 559 cells failed。
- 对于非单调信息警告，增加输出 when 信息，方便用户查看具体位置。

2.2.10 2025-02-25

- 重构命令行参数解析，使用 CLI11 库的子命令，在每个回调函数中实现 parse、mono 功能。

2.2.11 2025-02-26

- 子命令可自定义输出文件名、日志文件名，并设置了相应的默认值。
- 改为在 Logger 初始化时打印版本信息，避免在 -h，-v 时多余打印。
- LibFile::mono() 方法增加对 input_slew 的单调性检查，如果在输入时有指定，就检查 value 矩阵的每一列是否单调。经过测试，tcbn65lpbc.lib 输出与 liberate_lv 工具一致。

2.2.12 2025-02-27

- 多线程并行尝试，采用GDB 调试：
  - si2dr_liberty 库在解析 Liberty 文件时，可能使用了共享的数据结构（例如哈希表、字符串表）来存储解析结果或中间数据。
  - 在多线程并行解析多个文件时，不同的线程同时调用 si2dr_liberty 库的函数，并发地访问和操作这些共享数据结构。
  - si2dr_liberty 库可能没有采取足够的线程同步措施来保护这些共享数据结构，导致了数据竞争。
  - 数据竞争最终导致了内存损坏，使得 strcmp 函数在后续操作中访问了无效内存，触发了段错误。

2.2.13 2025-02-28

- 修改 LibFile 的构造函数与析构函数，read 方法改为私有，移入 parse 方法。将 si2dr 初始化与销毁放在 parse 方法中，使得 mono 方法与 si2dr 库解耦，从而实现多线程并行验证单调性。
- [x] 顶层 si2drPIGetGroups 未退出的 bug: WARNING: si2drPIQuit: GetGroups called 1 more times than IterQuit，内存泄漏？
- 学到了可以在 CMakeLists.txt 中使用 set(CMAKE_BUILD_TYPE Release) 减少编译器优化程度，配合GDB 等调试工具检查堆栈信息。

2.3 2025-03

2.3.1 2025-03-01

- sub_groups_iter 是在每次 for 循环的迭代中声明的局部变量，在每次迭代结束后其作用域结束，析构函数被调用，释放相关资源。所以 si2drPIQuit 未发现内存泄漏。
- 顶层 group_iter 的析构函数发生在 parse 方法结束时，所以 parse 方法末尾的 si2drPIQuit 检测到了顶层 groups 未退出的情况。
- 综上，si2drPIQuit 的警告是由于当时顶层 groups 未退出导致的，在 parse 方法结束后能正确释放，不会造成实际上的内存泄漏。
- [ ] 顺序执行 parse，log 输出仍存在问题：解析第二第三个库的时候会多余打印之前库的PVT信息，时间上也比单独解析一个库的总和要慢 (parse 6s 左右，mono 1s)。并行执行 mono，log 输出会集中到最后一个库的文件。
- 给 LibFile 类的私有属性添加了初始值，避免了未初始化的问题，吗？

2.3.2 2025-03-07

- 实现了 compare 子命令的解析，能够对两个库文件依次进行解析，并生成对应的 JSON 文件。
- [ ] 仍然存在日志重复输出的问题，多文件处理时数据可能会相互干扰。

2.3.3 2025-03-10

- 新增 supercell 子命令和 LibFile::supercell 方法，用于生成超级单元以供验证。
- 集成了 zlibboost 子命令，支持调用开源库特征化工具 zlibboost。已与开发者取得联系。
- 调整了子命令输出文件的位置为当前目录，避免文件混淆。
- 添加了 clear 子命令，方便清理生成的 JSON 文件和日志文件。

2.3.4 2025-03-14

- 添加了 tabulate/table 库，通过 CMake FetchContent 进行管理，用于生成格式化的表格输出。
- 新建了 compare.cpp 和 compare.hpp 文件，创建了 LibraryComparator 类，用于比较两个库文件的差异。目前实现了读取参考库 JSON 文件的功能。

2.3.5 2025-03-15

- 使用 `std::filesystem::path` 重构了 `LibFile` 类的文件路径管理，方便文件路径的拼接和处理。`LibFile` 类添加了成员变量 `basename_`、`filename_`、`libname_`、`jsonname_` 和 `loggername_`，用于存储文件名、库名、JSON 文件名和日志文件名。
- 通过作用域 (scope) 管理 `LibFile::parse` 方法中的顶层迭代器的生命周期，提前结束并调用 `si2drPIQuit` 保证资源释放，解决了分析多个库时日志重复输出和数据保存错误的问题。
- `clear` 子命令增加了删除 `.map` 和 `.md` 文件的功能。
- 实现了 `mono` 子命令的多线程并行。首先检查是否是多文件输入，如果是，就顺序检查是否准备好了每个文件的 JSON 文件，否则顺序进行多文件库解析。在所有文件的 JSON 准备就绪后，根据文件数量创建线程池，每个线程负责一个库的单调性检查。
- 重构了 `spdlog` 日志初始化，返回 logger 对象而不是设置全局默认 logger。保留名为 `APP_NAME` 的全局 logger，用于输出总体提示信息。
- 为 `LibFile` 类添加了独立的成员变量 logger，用于记录每个库文件的日志信息。
- 完成了 `supercell` 子命令的多线程并行，处理逻辑与 `mono` 子命令类似。
- `LibraryComparator::generateReport()` 方法测试了 `tabulate` 库的 markdown 表格输出功能，完善了报告的头部信息输出，包括参考库、比较库、相对容差、软件信息、报告生成时间和图例说明。

2.3.6 2025-03-16

- 针对 `compare` 子命令，增加了对报告文件名的检查，确保其以 `.md` 或 `.txt` 结尾。若不符合，则给出警告并自动添加 `.md` 后缀。
- 进一步完善了 `LibraryComparator::generateReport()` 方法。现在，该方法能够遍历比较库中的所有 cell，并在参考库中查找对应的 cell。如果找到，则调用 `LibraryComparator::compareCell()` 方法进行比较；否则，会记录 cell 未找到的警告信息。
- 新增了 `LibraryComparator::compareCell()` 方法，用于比较两个库中同名 cell 的输出引脚。该方法会遍历比较库 cell 的每个输出引脚，在参考库 cell 中寻找同名引脚。如果找到，则调用 `comparePin` 函数比较引脚；如果未找到，则记录警告信息。如果比较库 cell 没有输出引脚，则会记录一条信息级别的日志。
- 新增了 `LibraryComparator::comparePin()` 方法，用于比较两个库中同名 cell 的同名引脚。它会遍历比较库 pin 的每个 timing arc，并在参考库 pin 中查找相同类型的 timing arc。如果找到，则调用 `compareTimingArc` 函数进行比较；如果未找到，则记录警告信息。如果比较库 pin 没有 timing arc，则记录一条信息级别的日志。

- 修改了 LibraryComparator::compareTimingArc() 方法，用于比较两个 timing arc JSON 对象。该方法首先记录正在比较的 timing arc 的类型，然后提取 "related_pin" 属性。接着，它遍历预定义的 timing arc 名称列表 ("cell_rise", "cell_fall", "rise_transition", "fall_transition")，并在比较库中查找这些 timing arc。如果找到，则在参考库中查找对应的 timing arc，并调用 compareLut 方法进行比较。如果参考库中未找到对应的 timing arc，则记录警告信息。
- 修改了 LibraryComparator::compareLut() 方法 (原 compareValue 方法，已更正命名)，用于比较两个 JSON 对象中具有相同名称的 value。它首先提取两个 JSON 对象中的 "index_1" 和 "index_2" 数组，如果这两个数组不相等，则记录错误信息并返回。如果索引匹配，则比较 LUT 中的实际 value 值，并根据相对容差 (reltol) 和绝对容差 (abstol) 标记差异。
- 重新设计了层级比较方法，现在可以逐层级地传入 cell_name、pin_name、timing_type、related_pin 和 arc_name，以便在最内层制表或提示时能够准确输出层级信息。
- 修复了 LibraryComparator::compareValue() 方法的命名错误,已更正为 LibraryComparator::compareLut
- 已经可以输出表格，数据数量和商用工具结果一致，但格式还需要进一步调整。
- 新增了 abstol 参数，默认值为 0.002ns，用于设置绝对容差，与库验证工具保持一致。

2.3.7 2025-03-17

- 新增 verilog 和 spice 子命令，用于生成 Verilog 和 SPICE 代码。
- 完善 LibraryComparator 报告生成:
  - 为每个 cell 增加表头，仅在存在表格数据时输出表格，避免冗余输出。
  - 增加 cell 结果统计功能，输出结果统计表格，目前仅支持 Timing 中 Delay 的比较。

2.3.8 2025-03-18

- 优化统计表格，包含 | Cell Name | Data Type | Failed Count | Avg Diff | Avg Diff% | Max Diff | Max Diff% | Outliers |。
- 优化报告对比表格，包含 | Pin Name | Reference | Comparison | Diff | Diff % | Type | Arc Name | Row # | Index_1 | Column # | Index_2 | Note |
- 报告表格的 stdout 输出增加表头加粗、黄色的格式化。
- 增加 si2drSimpleAttrGetBooleanValue 封装，用于获取布尔类型的属性值，以判断引脚是否是时钟引脚。
- LibFile::mono() 增加对 min_pulse_width 的单调性检查:
  - 针对包含 "input_pins" 的 cell，遍历每个输入引脚。
  - 如果引脚是时钟引脚，则检查其 "timing_arcs"。

  - 针对 "timing_type" 为 "min_pulse_width" 的 timing arc，对 "rise_constraint" 和 "fall_↩ constraint" 调用 `checkTimingArcMonotonicity` 进行单调性检查。

- `checkTimingArcMonotonicity` 函数日志区分有无 when 信息，输出不同日志。
- `checkTimingArcMonotonicity` 核心比较功能取消了等于的情况，相等情况默认为通过。
- `checkTimingArcMonotonicity` 核心比较功能增加了判断 `related_pin` 是否等于当前 pin，不等于则跳过。最终改为：如果矩阵中当前的值小于前一个值，并且当前引脚不是相关引脚，或者当前值和前一个值都为零，那么就认为这些值不是单调递增的。

2.3.9 2025-03-19

- 完善 supercell 方法，如果有有时钟信号引脚，链式长度被设为 1，再生成超级单元。
- 解决 voltage 浮点数误差问题，std::round(voltage_ $*$ 100) / 100.0 保留两位小数。

2.3.10 2025-03-21

- 新增 func 子命令，用于检查库或者Verilog 文件的逻辑等价性。

2.3.11 2025-03-24

- 调研了 C++ 操作 Verilog 相关的库，找到了一个实用的 Verilog 库：`slang`，它可以解析出抽象语法树 (AST)。已修改 CMakeLists 文件，将 slang 集成到项目中，目前正在研究其 API。

2.3.12 2025-03-25

- 新增 `verilog_utils.cpp` 和 `verilog_utils.hpp` 文件，并创建了 `VerilogVisitor` 类，用于自定义遍历 Verilog AST。
- 验证了 slang 对不同类型逻辑单元的解析能力：
  - 简单组合逻辑（如 `AND2D0`）、复杂时序逻辑（如 `CMPE42D1`）和基本时序逻辑（如 `DFQD1`）均能被正确解析。
  - 用户自定义原语 (UDP)，例如 `tsmc_dff`，会被错误地识别为模块实例化，并产生 "Invalid instance declaration" 警告。
  - 成功提取了模块名、端口方向（`input`、`output`）及名称。
  - 能够解析命名端口连接的层次化实例化，并获取模块名、实例名称和端口映射关系。
  - 能够解析门级原语实例化，并获取门类型、实例名称、输出端口和输入端口。

2.3.13 2025-03-26

- 进一步完善了对 UDP（如 `tsmc_dff`）端口映射关系的解析，但其内部 Entry 尚未处理。
- 尝试了通过继承 `slang::syntax::SyntaxRewriter` 类创建自定义类 `CellExtractor`，实现了只保留目标模块、删除其他模块，并将修改后的语法树另存为新文件。使用 `slang::syntax::SyntaxPrinter::printFile()` 方法可以将 Verilog 代码输出到文件。
- 创建了 `CellPrinter` 类，继承自 `slang::syntax::SyntaxVisitor` 类。当访问到目标模块时，使用 `module.toString()` 方法也能将 Verilog 代码输出到文件，但发现输出结果中空行会被删除。
- 修改了 `LibFile::supercell()` 方法，该方法可以根据输入的 `cell_names` 生成指定的 supercell，并在遇到未找到的单元时提示警告信息。
- 优化了时钟引脚的处理方式，现在时钟引脚不再被视为 supercell 的输入引脚，而是仅记录为时序单元，并跳过存入 `input_pins` 集合的步骤。
- 引入了 Doxygen 文档生成工具，用于可视化分析项目，并生成了 Doxygen HTML 文档和 LaTeX 参考手册。
- [x] 尚未解决手册中文显示不正确的问题，可能需要自定义 LaTeX 头文件。

2.3.14 2025-03-27

- **重大失误**！误操作 Git 分支，导致部分代码修改丢失。务必牢记：** 及时提交代码！**
- 实现了 `LibFile::verilog` 方法的核心流程：
  - 首先调用 `this->supercell()` 生成超级单元。
  - 读取 `.map` 文件，提取单元映射关系到 `std::pair<std::string, std::string>` 中（原始单元名 -> 超级单元名/模块名）。
  - 从 `lib_json_` 中提取单元的输入/输出端口信息。
  - 根据是否存在时钟引脚，确定 Verilog 模块中 instance 的生成数量（时序逻辑为 1，组合逻辑等于 chain length）。
  - 通过字符串操作生成 ANSI 风格的端口列表，并与模块名组合成完整的 `fullModuleText`。
  - 使用 `slang::syntax::SyntaxTree::fromText` 从 `fullModuleText` 生成语法树。
  - 将语法树传递给 `ModuleRewriter` 类，使用其 `transform` 方法遍历模块成员，并通过 `handle()` 方法进行处理。
  - 最后，使用 `slang::syntax::SyntaxPrinter::printFile(*tree)` 将处理后的语法树输出到文件。
- `LibFile::verilog` 目前能够正确输出模块名和 ANSI 风格的端口声明。

2.3.15 2025-03-28

- 实现了在模块成员列表中插入新节点的功能。具体而言，在 ModuleRewriter::handle(const slang::syntax::ModuleDeclarationSyntax &module) 方法中，通过 insertAtBack(module.members, newDataNode) 函数，成功地在模块成员的末尾插入了链式单元所需的中间变量声明，例如 wire OP_i;。
- 成功地在模块内部添加了关键连接网络变量。通过解析 CELLNAME__X#__CRITICALPORT__OUTPUTPORT 格式的字符串，提取出关键输入端口（critical input port）和输出端口（OUTPUTPORT），用于后续的实例化端口连接。
- 增加了用于处理多输出接口的中间网络变量 P__UNUSEOUTPUTPORT，用于连接不需要考察的输出端口。
- 成功地在模块内添加了模块实例化所需的模块名和实例名称，完成了端口映射关系的连接处理。
- 验证了代码对简单组合逻辑单元（如 INVD0 反相器）和复杂组合逻辑单元（如 FA1D0 全加器）的输出能力，结果均符合预期。
- 为 ModuleRewriter 类添加了私有属性 std::map<std::string, std::string> portInfoMap_，用于存储端口映射关系，将端口名映射到其方向（input/output）。
- 完成了实例化端口连接的细节处理：
  - 如果是关键输入端口且为第一个实例，则直接连接到输入端口；否则，连接到 OP_(i-1) 这样的中间网络变量。
  - 如果是关键输出端口，且为最后一个实例，则连接到输出端口；否则，连接到 OP_i 中间网络变量。
  - 如果是其他输出端口且不是最后一个实例，则连接到 P_i__portName 这样的中间网络变量；否则，连接名等于端口名。
  - 其余输入端口，连接名等于端口名。
- 文档方面，GitHub 远程仓库已开放为公开访问，并配置了 GitHub Pages。可以通过 https://cedar17.github.io/ZlibValidation/ 访问项目文档，该页面包含 Doxygen 生成的 HTML 文档和仓库地址的链接。
- 配置了 GitHub Actions，实现了 Doxygen 文档和 Graphviz 图的自动化构建，以及 LaTeX 参考手册的自动编译（暂不支持中文）。每次在 dev 和 main 分支的提交都会触发文档和参考手册的生成，并发布到 gh-pages 分支。
- 将第三方库放置到 include_3rd_party 目录下，方便管理。修改了 CMakeLists 文件，将第三方库的头文件路径添加到 include_directories 中。修改了 Doxyfile 文件，将第三方库的头文件路径添加到 INPUT 中，便于文档生成工具理解第三方库函数、类的关系及使用方法。

2.3.16 2025-03-29

- 修复了 Doxygen 生成 LaTeX 参考手册时中文显示不正确的问题。通过修改 Doxyfile 文件，添加 `EXTRA_PACKAGES = ctex` 选项，并指定 `lualatex` 作为编译器，成功解决了中文显示问题。同时，从 `include_3rd_party` 目录中移除了 `Allsyntax.h` 文件，避免了由于 LaTeX 文档过大导致的 "TeX capacity exceeded, sorry [main memory size=5000000]" 内存不足问题。
- 实现了 Verilog 文件的顶层模块生成功能，采用非 ANSI 风格的端口声明方式，格式与 `liberate_lv` 工具生成的完全一致。实现过程如下：
  - 首先将各模块的 Verilog 代码存放到临时文件中。
  - 在处理模块的过程中，使用 `std::vector<std::string>` 收集每个模块的 `input_pins` 和 `output_pins`。
  - 遍历所有模块名称列表，拼接模块名和引脚名，生成顶层模块的 `all_input_ports` 和 `all_output_ports`。
  - 使用字符串拼接方式构建 `validate_top` 模块的完整代码。
  - 最后将临时文件内容与顶层模块代码合并，输出到最终的 Verilog 文件中。
- 更改 `deploy-docs.yml`，修复了 GitHub Action 生成的 LaTeX 参考手册中的图像显示问题。具体步骤如下：
  - 设置系统时区为 Asia/Shanghai (东八区)。
  - 安装 miniconda 并更新 conda。
  - 通过 conda 安装 1.9.5 版本的 Doxygen，并安装 graphviz 以解决依赖关系，确保每个 dot 图都能正确编译出 pdf 文件。
  - 使用 `doxygen` 命令生成 HTML 文档和 tex 文件。
  - 采用 action marketplace 提供的 `xu-cheng/texlive-action@v2` 安装全量的 texlive2024。
  - 进入 `./doc_doxygen/latex` 目录，运行 `make` 命令使用 lualatex 编译 tex 文件，生成 pdf 文件。
  - 使用 `JamesIves/github-pages-deploy-action@v4` 将生成的 pdf 文件上传到 GitHub Pages 发布，并保留 `README.md` 和 `index.html` 文件。

2.3.17 2025-03-30

- 完成了 `LibFile::logic(const std::string &cell_name)` 方法，该方法返回指定 cell_name 所有输出引脚对应的逻辑表达式，以 `std::map<std::string, std::string>` 的形式返回，存储输出引脚名称到其逻辑表达式字符串的映射关系.
- 实现了 `func` 子命令的基本框架，用于检查库或 Verilog 文件的逻辑等价性。
  - 完善了 `funcLibFile` 函数，能够处理 Liberty 文件和 Verilog 文件作为参考和比较对象。
  - 增加了文件类型判断，根据文件扩展名选择不同的处理方式。

- 实现了对 .lib 文件的解析，并提取指定 cell 的逻辑表达式。
- 目前仅支持 Liberty 文件的逻辑表达式提取，Verilog 文件的逻辑表达式提取功能尚未完成 (TODO)。
- 增加了对 report 文件名的检查，确保其以 .md 或 .txt 结尾。若不符合，则给出警告并自动添加 .md 后缀。
- 实现了从 Liberty 文件中提取逻辑表达式的功能，使用了 LibFile::logic 方法。
- 增加了日志输出，记录每个 cell 的逻辑表达式提取和比较过程。
- 逻辑比较部分目前为空，待实现 (TODO)。

2.3.18 2025-03-31

- 实现了从 Verilog 结构网表中提取逻辑表达式的功能，使用了 LogicExtractor 类。
  - 设计了如何使用 Slang 库从 Verilog 结构网表中提取逻辑表达式的方案:
    - 使用 Slang 解析 Verilog 文件获取语法树 (AST)。
    - 遍历 AST 定位目标模块。
    - 在目标模块内构建网表的内部表示，识别输入、输出、线网及门级连接关系。
    - 对每个输出端口，递归反向追踪门和线网直至到达主输入端口。
    - 在回溯过程中根据遇到的门类型构建逻辑表达式。
    - 应用记忆化技术避免对相同线网/信号的重复计算。
  - 创建了 LogicExtractor 类，用于从 Verilog 代码中提取逻辑表达式。
    - LogicExtractor 类使用 Slang 库解析 Verilog 文件，并构建目标模块的网表表示。
    - 实现了 handle(const slang::syntax::ModuleDeclarationSyntax &module) 方法，用于定位目标模块，并初始化内部数据结构。
    - 实现了 handle(const slang::syntax::PortDeclarationSyntax &portDecl) 方法，用于提取端口信息，包括端口方向和名称。
    - 实现了 handle(const slang::syntax::NetDeclarationSyntax &netDecl) 方法，用于提取线网信息。
    - 实现了 handle(const slang::syntax::PrimitiveInstantiationSyntax &primitiveInst) 方法，用于提取门级单元信息，包括门类型、输入和输出信号。
    - 实现了 getLogicExpressions() 方法，用于根据提取的网表信息，递归地推导每个输出端口的逻辑表达式。
    - 实现了 deriveLogicRecursive(const std::string &signalName) 方法，用于递归地推导指定信号的逻辑表达式。
    - 实现了 formatExpression(const GateInfo &gateInfo, const std::vector<std::string> &inputExprs) 方法，用于根据门类型和输入表达式，生成逻辑表达式字符串。
  - 实现了 extractAndPrintNetlistInfo(const std::string &verilog_file, const std::string &cell) 函数，用于提取并打印网表信息，包括输入、输出、线网和门级单元。

  - 实现了 `extractLogicFromVerilog(const std::string &verilog_file, const std::string &cell)` 函数，用于从 Verilog 文件中提取指定 cell 的逻辑表达式，并以 `std::map<std::string, std::string>` 的形式返回。

- 为了更好地组织代码结构，将 LogicExtractor 类从 verilog_utils.hpp 和 verilog_utils.cpp 中分离出来，并创建了独立的 LogicExtractor.hpp 和 LogicExtractor.cpp 文件。同时，统一了所有 hpp 和 cpp 文件的命名规范，使其与对应的类名保持一致。
- 新增 LogicComparator 类及其头文件、源文件，用于比较两个逻辑表达式的等价性，并生成比较报告。

2.4 2025-04

2.4.1 2025-04-01

- 整理头文件引用，删除了不必要的引用，提高编译效率。
- 引入 `exprtk.cpp` 到第三方头文件目录，为后续的表达式解析做准备。
- 完善了 LogicComparator 类的实现，增加了其示例代码模板 `template <typename T> void logic();`，该模板能够对一个表达式遍历输入变量的所有组合，计算出逻辑值，并打印真值表，为逻辑等价性验证提供基础。
- 实现了 LogicComparator::preprocessExpression() 方法，用于预处理逻辑表达式，目的是简化表达式，使其更易于比较。预处理的逻辑如下：
  - ** 统一 AND 处理 **：将 * 和空格隐式 AND 都转换为带空格的 `and`，保证 AND 运算符的显式性。
  - ** 显式符号处理 **：
    - 在所有括号 () 周围添加空格，方便后续分词。
    - 将 + 替换为 `or`。
    - 将 ^ 替换为 `xor`。
    - 将 ! (非 != 中的 !) 替换为 `not`。
    - 将 * 替换为 `and`。
  - ** 分词 **：基于空格将处理后的字符串分割为 token 列表。
  - ** 精确插入隐式 AND**：
    - 遍历 token 列表。
    - 检查当前 token 和下一个 token，如果满足以下条件，则在当前 token 之后插入 `and`：
      - 当前 token 是一个标识符 (如 `A`, `CIX`) 或右括号)。
      - 下一个 token 是一个标识符或左括号 (。
      - 当前 token 不是一个显式逻辑运算符 (`and`, `or`, `xor`, `not`) 或左括号 (。
      - 下一个 token 不是一个显式逻辑运算符或右括号)。

  - ** 重组与清理 **：将处理后的 token 列表用单个空格连接起来，并进行最终的空格清理（去除多余空格，修剪首尾空格），得到最终的预处理表达式。

- 在 CMakeLists.txt 中，强制启用了 lld 链接器和 ccache，以提升链接和编译速度。
- 为了保持 main.cpp 的整洁，将 LibFile() 函数的相关操作剥离出来，并放入 LibFileOperations.cpp 文件中。
- 实现了 LogicComparator::extractVariables() 方法，用于从两个表达式中提取并验证变量名，确保它们一致。
  - 该方法使用正则表达式初步提取所有可能的标识符。
  - 对每个提取出的标识符，先用 isIdentifier 函数（检查大写开头等）进行验证。
  - 通过 isIdentifier 验证后，仍然将其转换为小写并与 keywords 集合比较，以过滤掉可能的关键词（这是一个保守但安全的操作）。
  - 只有通过这两步验证的标识符才被认为是有效变量，并以原始大小写形式添加到各自的 std::set 中。
  - 最后比较两个 std::set 是否相等，如果不等则报告差异并返回 false，如果相等则将变量列表（保留原始大小写）排序后存入 sorted_vars 并返回 true。
- 修复了 LogicComparator::preprocessExpression() 方法中对 not 操作符的处理。exprtk 库要求 not 关键字后必须跟着括号，如 not(A) 而非 not A。为此增加了专门的处理步骤：
  - 遍历预处理后的 token 列表，识别所有 not 关键字。
  - 当发现 not 后，检查其后是否跟随标识符（通过 isIdentifier() 函数判断）。
  - 如果后接标识符，自动将 not A 形式转换为 not(A)，具体做法是依次添加 not、(、标识符和)，并跳过已处理的标识符。
  - 如果后接其他元素（如 (、其他运算符或表达式结尾），保留原样，让 ExprTk 处理 not(表达式) 形式。
  - 对其他 token，直接添加到最终 token 列表，不做特殊处理。
- 实现了 LogicComparator::compareSingleExpressionPair() 方法，用于比较两个逻辑表达式的等价性：
  - 接收预处理后的表达式字符串和已排序的变量列表作为输入参数。
  - 为参考表达式和比较表达式分别创建 ExprTk 环境（符号表、表达式对象和解析器）。
  - 将所有变量添加到两个符号表中，确保变量一致性和顺序性。
  - 使用 ExprTk 解析器编译两个表达式，并检查可能的语法错误。
  - 通过二进制计数法，遍历所有可能的输入变量组合（2^N 种），其中 N 为变量数量。
  - 对每种组合：
    * 设置两个符号表中的变量值（0 或 1）。
    * 计算两个表达式的值，并处理可能的运行时错误。
    * 将计算结果转换为布尔值并存入结果向量。

    * 同步构建真值表，记录输入组合和对应的输出值。
  - 比较两个结果向量判断表达式是否等价。
  - 将生成的真值表存入 PinComparisonResult 对象的 std::optional 成员中，以便后续报告生成。

- 实现了 template <typename T> std::map<std::string, PinComparisonResult> LogicComparator::compareCellLogic();
  - 该方法用于比较两个 cell 的逻辑表达式，返回一个 std::map，键为 cell 名称，值为 PinComparisonResult 结构体。
  - 遍历在类私有属性里的整个独立输出引脚键，提取其值作为逻辑字符串表达式，首先使用 preprocessExpression 进行预处理，然后使用 extractVariables 提取变量向量。
  - 调用 LogicComparator::compareSingleExpressionPair() 方法进行比较，结果存入 PinComparisonResult 对象。
  - 最后将 PinComparisonResult 对象存入 std::map 中，键为 cell 名称，值为 PinComparisonResult 对象，返回。
- 实现了 LogicComparator::generateReport() 方法，报告格式仍需要调整。

2.4.2 2025-04-02

- 取消所有模板函数的 template <typename T> 声明，改为使用 double 作为参数类型，加快编译速度。
- 完善了 LogicComparator::generateReport() 方法，增加了对逻辑比较结果的详细报告输出，包括：
  - 元数据，例如：**Performed by ZlibValidation v0.1.0 from Song Zixuan. on: Wed Apr 2 11:11:39 2025**
  - 图例，包括参考 pin 名称、pin function，参考 pin 名称、pin function，符号解释表。
  - 参考 pin 的真值表
  - 比较 pin 的真值表
  - 比较结果表格，例如：

Property	Value
Status	[OK]
Reference (Raw)	(!((!(A1+A2))+B))
Comparison (Raw)	!((!A1 * !A2) + B)
Reference (Processed)	(not((not(A1 or A2)) or B))
Comparison (Processed)	not((not(A1) and not(A2)) or B)
Ref Expression Compiled	Yes
Comp Expression Compiled	Yes
Logically Equivalent	Yes

2.4.3 2025-04-05

- 新增了 LibFile::spice() 方法，为 spice 子命令添加了 SPICE 网表生成功能。 - 通过执行 which v2lvs 命令检测系统中是否安装了 V2LVS 工具。如果已安装，则调用该工具生成基本的 SPICE 网表。 - 添加了 --vl 和 --sl 命令行选项，允许用户指定 Verilog 库文件和 SPICE 库文件的路径。 - 目前已能生成基本的 SPICE 网表，后续需要进一步完善输出格式。

```
Generate SPICE file for given Liberty file
Usage: ./zlibvalidation spice [OPTIONS] library_path...
Positionals:
  library_path TEXT:FILE ... REQUIRED
                            Specify the library file to process
Options:
  -h,--help                 Print this help message and exit
  -l,--log TEXT             Specify the log file name. Default: <basename>.spice.log
  -c,--chain INT            Specify the chain length for SPICE generation. Default: 1
  --cells TEXT ...          Specify the cell names to generate SPICE for
  --vl TEXT                 Specify the location of the Verilog primitive library file
  --sl TEXT                 Specify the location of the SPICE library file to be included in the output
```

- 实现私有方法 LibFile::splitString()，用于将字符串按空格分割成多个子字符串，并返回一个 std::vector<std::string>。
- 实现了 LibFile::generateRCLines() 方法，用于为给定的 net 生成 RC lines。
  - 该方法根据 isFinalStage 参数，生成不同的 RC 结构。
  - 如果 isFinalStage 为 false，则生成 R1-C1-R2 结构。
  - 如果 isFinalStage 为 true，则生成 R1-C1 结构。 - RC 的默认值参考目标 SPICE 文件。
- 实现了 LibFile::modifySpiceNetlist() 方法，用于修改 v2lvs 生成的 SPICE 网表。
  - 该方法添加元数据注释，包括应用名称、版本、作者和时间戳。
  - 在第一个 .INCLUDE 指令后插入指定的 global line。
  - 修改 subcircuit，在 .SUBCKT 定义的端口列表中添加 VDD VSS。
  - 对于 subcircuit 中的每个 instance line(以 "X' 或'x' 开头)，修改输出引脚名称，添加 VDD/↩ VSS 连接，并调用 generateRCLines 生成 R/C lines。
  - 保留原始注释，跳过空行、v2lvsheader line 和原始的.GLOBALline。
- 完全实现了 LibFile::spice()` 方法，能够生成 SPICE 网表并输出到指定文件中。SPICE 网表的生成过程如下:
  - 检查是否指定了 Verilog 库文件和 SPICE 库文件，如果没有指定，则使用默认路径。
  - 调用 v2lvs 工具生成初步的 SPICE 网表。
  - 调用 LibFile::generateRCLines() 方法，为 SPICE 网表中的每个 instance 生成对应的 RC lines。
  - 调用 LibFile::modifySpiceNetlist() 方法，修改 SPICE 网表，添加元数据、插入 global line、调整 instance lines，并生成最终的 SPICE 网表。
  - 将最终的 SPICE 网表输出到指定文件中。

2.4.4 2025-04-07

- 将项目研发日记从 README.md 迁移至独立的 ChangeLog.md 文件，存放于 doc/ 目录下，集中记录更新与变更历史。
- 全面完善 README.md，补充项目描述、目录、动机、安装指南、使用示例、文档与手册、致谢等内容，提升项目信息的完整性。
- 调整 CMakeLists.txt，取消强制使用 ccache 和 lld 的设定，改为检测系统环境，仅在存在相应工具时启用，增强项目兼容性。
- 移除 FetchContent 中的 QUIET 选项，增加必要提示信息，更清晰地展示依赖库的获取状态。
- 项目版本更新至 v1.1.0。
- 明确 CMake 版本需求为 (VERSION 3.21...3.31)，确保项目在指定版本范围内构建。
- 在 README.md 中，特别感谢 ZlibBoost 项目，并新增编译文档和参考手册的教程，同时将 ccache 和 lld 列为可选编译准备。

2.4.5 2025-04-10

- 修正了 printInfo() 函数中的注释错误，将文件日志级别更正为 trace，而非 debug。
- 添加了 mono 子命令的测试命令到 test.sh 脚本中。

2.4.6 2025-04-15

- 修复了 LibraryComparator::generateReport 方法中的一个 bug，max_diff 现在能基于绝对值比较得出最大差异，并且最大差异百分比保持一致。
- 完善了所有关键子命令的测试命令，确保每个子命令都能在 test.sh 脚本中正确执行。

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Class Documentation

6.1 AttributesIterator Class Reference

`#include <Iterators.hpp>`

Public Member Functions

- AttributesIterator (si2drAttrsIdT attrs, si2drErrorT &err)
- ~AttributesIterator ()
- void next ()
- bool end ()
- LibAttribute get ()

Private Attributes

- si2drAttrsIdT attrs_
- si2drAttrIdT attr_
- si2drErrorT & err_

6.1.1 Detailed Description

Definition at line 23 of file Iterators.hpp.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 AttributesIterator()

```
AttributesIterator::AttributesIterator (
                si2drAttrsIdT attrs,
                si2drErrorT & err )
```

Definition at line 14 of file Iterators.cpp.

6.1.2.2 ~AttributesIterator()

```
AttributesIterator::~AttributesIterator ( )
```

Definition at line 18 of file Iterators.cpp.

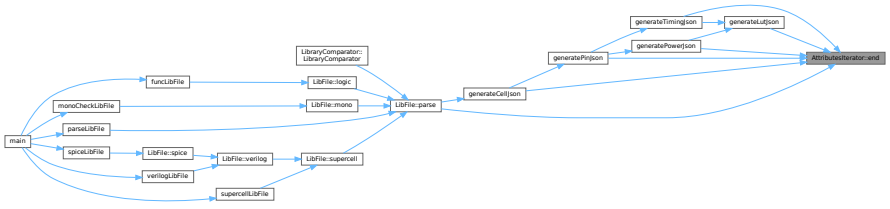
6.1.3 Member Function Documentation

6.1.3.1 end()

```
bool AttributesIterator::end ( )
```

Definition at line 21 of file Iterators.cpp.

Here is the caller graph for this function:

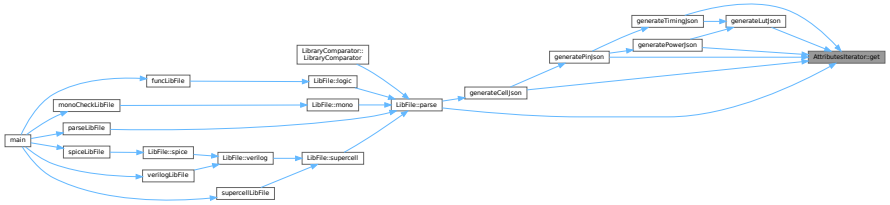


6.1.3.2 get()

```
LibAttribute AttributesIterator::get ( )
```

Definition at line 23 of file Iterators.cpp.

Here is the caller graph for this function:

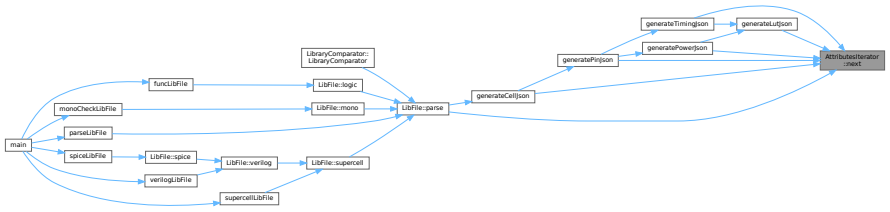


6.1.3.3 next()

```
void AttributesIterator::next ( )
```

Definition at line 20 of file Iterators.cpp.

Here is the caller graph for this function:



6.1.4 Member Data Documentation

6.1.4.1 attr_

```
si2drAttrIdT AttributesIterator::attr_  [private]
```

Definition at line 33 of file Iterators.hpp.

6.1.4.2 attrs_

```
si2drAttrsIdT AttributesIterator::attrs_  [private]
```

Definition at line 32 of file Iterators.hpp.

6.1.4.3 err_

```
si2drErrorT& AttributesIterator::err_  [private]
```

Definition at line 34 of file Iterators.hpp.

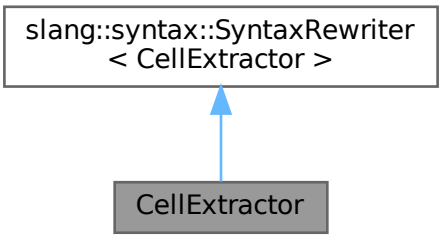
The documentation for this class was generated from the following files:

- include/Iterators.hpp
- src/Iterators.cpp

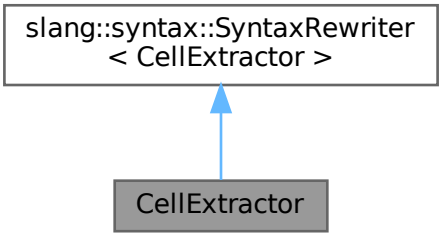
6.2 CellExtractor Class Reference

```
#include <verilog_utils.hpp>
```

Inheritance diagram for CellExtractor:



Collaboration diagram for CellExtractor:



Public Member Functions

- CellExtractor (const std::string &targetCell)
- void handle (const slang::syntax::ModuleDeclarationSyntax &module)
- bool foundTargetCell () const

Private Attributes

- const std::string & targetCell_
- bool foundTarget_

6.2.1 Detailed Description

Definition at line 33 of file verilog_utils.hpp.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 CellExtractor()

```
CellExtractor::CellExtractor (
            const std::string & targetCell )  [inline], [explicit]
```

Definition at line 35 of file verilog_utils.hpp.

6.2.3 Member Function Documentation

6.2.3.1 foundTargetCell()

```
bool CellExtractor::foundTargetCell ( ) const
```

Definition at line 364 of file verilog_utils.cpp.

Here is the caller graph for this function:

6.2.3.2 handle()

```
void CellExtractor::handle (
                const slang::syntax::ModuleDeclarationSyntax & module )
```

Definition at line 348 of file verilog_utils.cpp.

6.2.4 Member Data Documentation

6.2.4.1 foundTarget_

```
bool CellExtractor::foundTarget_  [private]
```

Definition at line 42 of file verilog_utils.hpp.

6.2.4.2 targetCell_

```
const std::string& CellExtractor::targetCell_  [private]
```

Definition at line 41 of file verilog_utils.hpp.

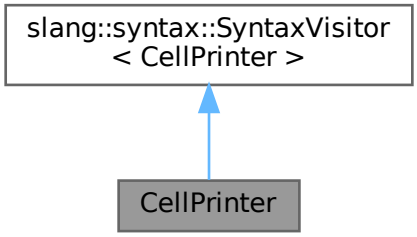
The documentation for this class was generated from the following files:

- include/verilog_utils.hpp
- src/verilog_utils.cpp

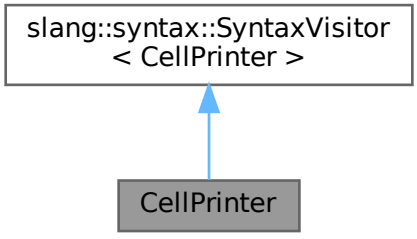
6.3 CellPrinter Class Reference

```
#include <verilog_utils.hpp>
```

Inheritance diagram for CellPrinter:



Collaboration diagram for CellPrinter:



Public Member Functions

- CellPrinter (const std::string &targetCell, std::ostream &out)
- void handle (const slang::syntax::ModuleDeclarationSyntax &module)

Private Attributes

- const std::string & targetCell_
- std::ostream & out_
- bool foundTarget_

6.3.1 Detailed Description

Definition at line 46 of file verilog_utils.hpp.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 CellPrinter()

```
CellPrinter::CellPrinter (
                const std::string & targetCell,
                std::ostream & out )  [inline], [explicit]
```

Definition at line 48 of file verilog_utils.hpp.

6.3.3 Member Function Documentation

6.3.3.1 handle()

```
void CellPrinter::handle (
                const slang::syntax::ModuleDeclarationSyntax & module )
```

Definition at line 367 of file verilog_utils.cpp.

6.3.4 Member Data Documentation

6.3.4.1 foundTarget_

```
bool CellPrinter::foundTarget_  [private]
```

Definition at line 55 of file verilog_utils.hpp.

6.3.4.2 out_

```
std::ostream& CellPrinter::out_  [private]
```

Definition at line 54 of file verilog_utils.hpp.

6.3.4.3 targetCell_

```
const std::string& CellPrinter::targetCell_  [private]
```

Definition at line 53 of file verilog_utils.hpp.

The documentation for this class was generated from the following files:

- include/verilog_utils.hpp
- src/verilog_utils.cpp

6.4 GateInfo Struct Reference

```
#include <LogicExtractor.hpp>
```

Public Attributes

- slang::parsing::TokenKind kind
- std::string gateTypeName
- std::vector< std::string > inputSignals
- std::string outputSignal

6.4.1 Detailed Description

Definition at line 9 of file LogicExtractor.hpp.

6.4.2 Member Data Documentation

6.4.2.1 gateTypeName

```
std::string GateInfo::gateTypeName
```

Definition at line 11 of file LogicExtractor.hpp.

6.4.2.2 inputSignals

```
std::vector<std::string> GateInfo::inputSignals
```

Definition at line 12 of file LogicExtractor.hpp.

6.4.2.3 kind

```
slang::parsing::TokenKind GateInfo::kind
```

Definition at line 10 of file LogicExtractor.hpp.

6.4.2.4 outputSignal

```
std::string GateInfo::outputSignal
```

Definition at line 13 of file LogicExtractor.hpp.

The documentation for this struct was generated from the following file:

- include/LogicExtractor.hpp

6.5 GroupsIterator Class Reference

`#include <Iterators.hpp>`

Public Member Functions

- GroupsIterator (si2drGroupsIdT groups, si2drErrorT &err)
- ~GroupsIterator ()
- void next ()
- bool end ()
- LibGroup get ()

Private Attributes

- si2drGroupsIdT groups_
- si2drGroupIdT group_
- si2drErrorT & err_

6.5.1 Detailed Description

Definition at line 9 of file Iterators.hpp.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 GroupsIterator()

```
GroupsIterator::GroupsIterator (
                si2drGroupsIdT groups,
                si2drErrorT & err )
```

Definition at line 3 of file Iterators.cpp.

6.5.2.2 ~GroupsIterator()

```
GroupsIterator::~GroupsIterator ( )
```

Definition at line 7 of file Iterators.cpp.

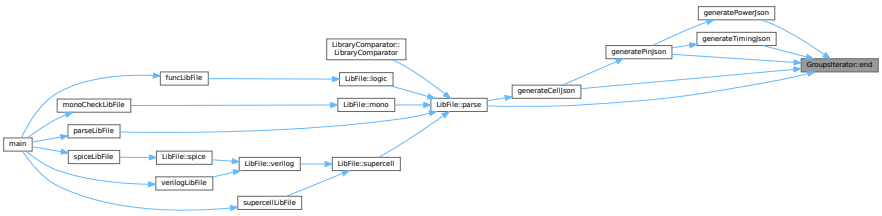
6.5.3 Member Function Documentation

6.5.3.1 end()

```
bool GroupsIterator::end ( )
```

Definition at line 10 of file Iterators.cpp.

Here is the caller graph for this function:

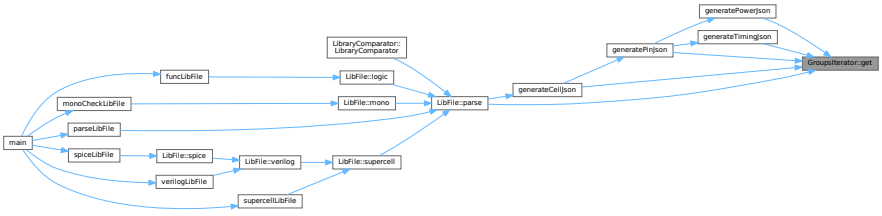


6.5.3.2 get()

```
LibGroup GroupsIterator::get ( )
```

Definition at line 12 of file Iterators.cpp.

Here is the caller graph for this function:

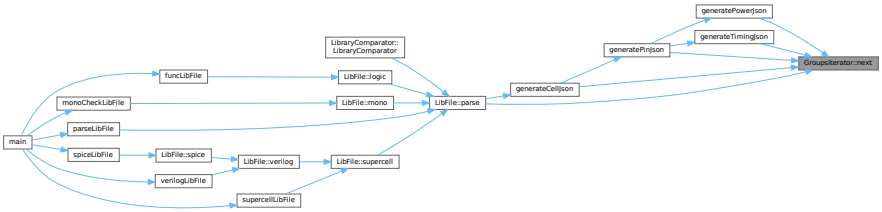


6.5.3.3 next()

```
void GroupsIterator::next ( )
```

Definition at line 9 of file Iterators.cpp.

Here is the caller graph for this function:



6.5.4 Member Data Documentation

6.5.4.1 err_

```
si2drErrorT& GroupsIterator::err_  [private]
```

Definition at line 20 of file Iterators.hpp.

6.5.4.2 group_

```
si2drGroupIdT GroupsIterator::group_  [private]
```

Definition at line 19 of file Iterators.hpp.

6.5.4.3 groups_

```
si2drGroupsIdT GroupsIterator::groups_  [private]
```

Definition at line 18 of file Iterators.hpp.

The documentation for this class was generated from the following files:

- include/Iterators.hpp
- src/Iterators.cpp

6.6 LibAttribute Class Reference

`#include <LibAttribute.hpp>`

Public Member Functions

- LibAttribute (si2drAttrIdT attr, si2drErrorT &err)
- ~LibAttribute ()
- std::string getName ()
- bool isComplex ()
- si2drValuesIdT getValues ()
- long int getInt ()
- double getFloat ()
- std::string getString ()
- bool getBoolean ()

Private Attributes

- si2drAttrIdT attr_
- si2drErrorT & err_

6.6.1 Detailed Description

Definition at line 8 of file LibAttribute.hpp.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 LibAttribute()

```
LibAttribute::LibAttribute (
                si2drAttrIdT attr,
                si2drErrorT & err )
```

Definition at line 3 of file LibAtrribute.cpp.

6.6.2.2 ~LibAttribute()

```
LibAttribute::~LibAttribute ( )
```

Definition at line 5 of file LibAtrribute.cpp.

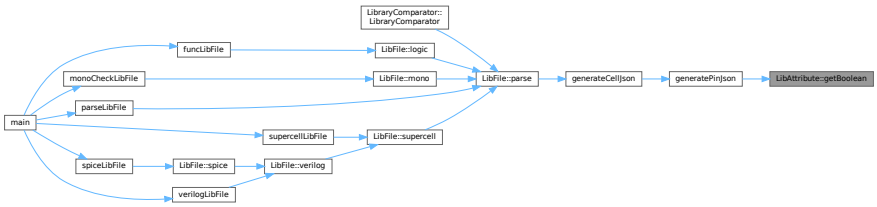
6.6.3 Member Function Documentation

6.6.3.1 getBoolean()

```
bool LibAttribute::getBoolean ( )
```

Definition at line 25 of file LibAtrribute.cpp.

Here is the caller graph for this function:

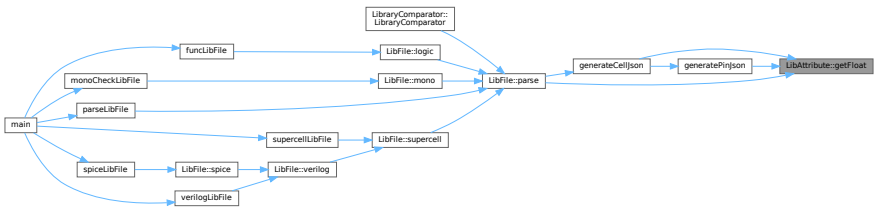


6.6.3.2 getFloat()

```
double LibAttribute::getFloat ( )
```

Definition at line 18 of file LibAtrribute.cpp.

Here is the caller graph for this function:

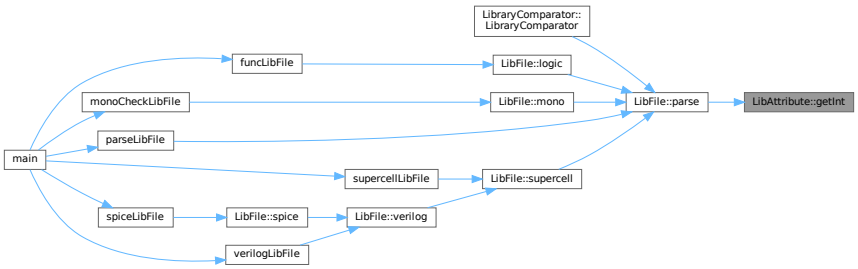


6.6.3.3 getInt()

```
long int LibAttribute::getInt ( )
```

Definition at line 16 of file LibAtrribute.cpp.

Here is the caller graph for this function:

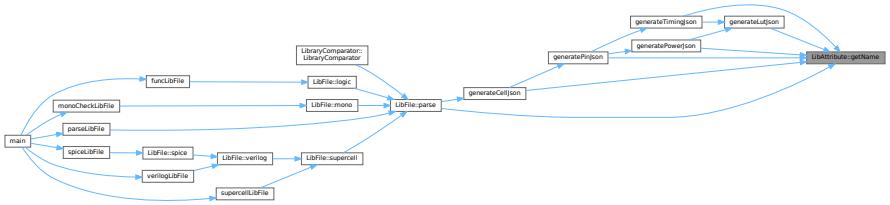


6.6.3.4 getName()

```
std::string LibAttribute::getName ( )
```

Definition at line 7 of file LibAtrribute.cpp.

Here is the caller graph for this function:

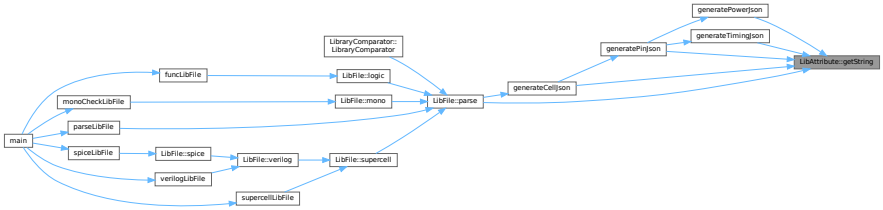


6.6.3.5 getString()

```
std::string LibAttribute::getString ( )
```

Definition at line 20 of file LibAtrribute.cpp.

Here is the caller graph for this function:

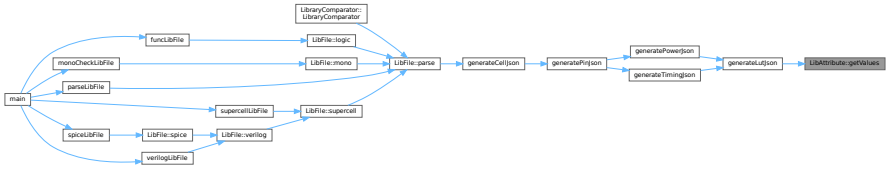


6.6.3.6 getValues()

```
si2drValuesIdT LibAttribute::getValues ( )
```

Definition at line 14 of file LibAtrribute.cpp.

Here is the caller graph for this function:



6.6.3.7 isComplex()

```
bool LibAttribute::isComplex ( )
```

Definition at line 12 of file LibAtrribute.cpp.

6.6.4 Member Data Documentation

6.6.4.1 attr_

```
si2drAttrIdT LibAttribute::attr_  [private]
```

Definition at line 21 of file LibAttribute.hpp.

6.6.4.2 err_

```
si2drErrorT& LibAttribute::err_  [private]
```

Definition at line 22 of file LibAttribute.hpp.

The documentation for this class was generated from the following files:

- include/LibAttribute.hpp
- src/LibAtrribute.cpp

6.7 LibFile Class Reference

```
#include <LibFile.hpp>
```

Public Member Functions

- LibFile (const std::string &filepath, const std::string &loggername)

  Constructs a LibFile object with specified file path and logger name.
- ~LibFile ()
- void writeJsonToFile ()

  Writes the JSON data stored in the object to a file.
- void parse ()

  Parses the Liberty file and extracts library information into a JSON structure.
- void modify ()
- void mono (const bool is_slew)

  Validates that lookup tables are monotonically increasing with respect to the output load.
- void supercell (const int chain_length, const std::vector< std::string > &cell_names)

Creates supercells based on standard cells in the library file.

- void verilog (const int chain_length, const std::vector< std::string > &cell_names)

  Generates Verilog files for cell validation based on the library file.

- void spice (const int chain_length, const std::vector< std::string > &cell_names, const std::string &verilog_lib_file, const std::string &spice_lib_file)

  Generates a SPICE netlist for the library based on a temporary Verilog file, using the V2LVS tool.

- std::map< std::string, std::string > logic (const std::string &cell_name)

  Retrieves the logic functions for the output pins of a specified cell within the Liberty file.

Public Attributes

- std::shared_ptr< spdlog::logger > logger_
- std::filesystem::path filepath_
- std::string basename_
- std::string filename_
- std::string libname_ = ""
- std::string jsonname_ = ""
- std::string loggername_ = ""
- json lib_json_ = json::object()

Private Member Functions

- void read ()

  Reads the Liberty file specified by the filename_ member variable.

- bool checkTimingArcMonotonicity (const json &cell, const json &pin, const json &arc, const std::string &type, bool is_slew)

  Checks if the values in a timing arc are monotonically increasing.

- std::vector< std::string > splitString (const std::string &s)

  Splits a string into a vector of tokens, using whitespace as the delimiter.

- void generateRCLines (std::ofstream &outFile, const std::string &netName, int instanceIndex, bool isFinalStage)

  Generates RC lines for a given net in either an intermediate or final stage.

- bool modifySpiceNetlist (const std::string &v2lvsSpiceFile, const std::string &finalSpiceFile, const std::string &targetGlobalLine)

  Modifies a SPICE netlist generated by v2lvs, adding metadata, inserting a global line, and adjusting instance lines within subcircuits to include VDD/VSS connections and generate corresponding R/C lines.

Private Attributes

- si2drErrorT err_
- int process_ = 0
- float voltage_ = 0.0
- int temperature_ = 0

6.7.1 Detailed Description

Definition at line 23 of file LibFile.hpp.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 LibFile()

```
LibFile::LibFile (
                const std::string & filepath,
                const std::string & loggername )
```

Constructs a LibFile object with specified file path and logger name.

This constructor initializes a LibFile object with the given file path and creates a logger with the specified name. It sets up:

- Two logging sinks: a colored console sink and a file sink
- File path information including filename, basename, and a corresponding JSON filename
- Log levels (info for console, debug for file)

Parameters

filepath	The path to the file to be processed
loggername	The name for the logger and the log file

Definition at line 15 of file LibFile.cpp.

6.7.2.2 ~LibFile()

```
LibFile::~LibFile ( )
```

Definition at line 35 of file LibFile.cpp.

6.7.3 Member Function Documentation

6.7.3.1 checkTimingArcMonotonicity()

```
bool LibFile::checkTimingArcMonotonicity (
                const json & cell,
                const json & pin,
                const json & arc,
                const std::string & timing_arc_name,
                bool is_slew )  [private]
```

Checks if the values in a timing arc are monotonically increasing.

This function examines the values in the specified timing arc to ensure they are monotonically increasing. It checks monotonicity in two dimensions:

1. Across rows (by output load capacitance)
2. Across columns (by input slew, only if is_slew is true)

The function logs detailed information about any non-monotonic values found, including cell name, pin names, and conditional statements (when clause) if present.

Parameters

cell	The JSON object representing the cell being checked
pin	The JSON object representing the pin being checked
arc	The JSON object representing the timing arc being checked
timing_arc_name	The name of the timing arc to check (e.g., "cell_rise", "cell_fall")
is_slew	Boolean flag indicating whether to check monotonicity across input slew values

Returns

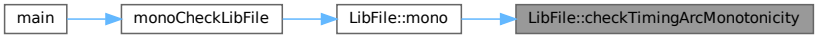
true if all values in the timing arc are monotonic, false otherwise

Exceptions

None,but	logs errors or warnings for invalid data formats or non-monotonic values

Definition at line 205 of file LibFile.cpp.

Here is the caller graph for this function:



6.7.3.2 generateRCLines()

```
void LibFile::generateRCLines (
                std::ofstream & outFile,
                const std::string & netName,
                int instanceIndex,
                bool isFinalStage )  [private]
```

Generates RC lines for a given net in either an intermediate or final stage.

This function generates SPICE-like resistor-capacitor (RC) lines and writes them to the provided output file stream. The generated RC structure differs based on whether it's an intermediate stage or the final stage. In the intermediate stage, an R1-C1-R2 structure is created. In the final stage, a simplified R1-C1 structure is used.

Parameters

outFile	The output file stream to write the RC lines to.
netName	The name of the net to generate RC lines for. This name is used to construct node names.
instanceIndex	A unique index for the instance, used to create unique component names (R1, C1, R2).
isFinalStage	A boolean flag indicating whether this is the final stage. If true, a simplified RC structure is generated. If false, an intermediate RCR structure is generated.

- The function uses predefined default RC values (R1, C1, R2) that are specific to either the intermediate or final stage.
- The `CAP_GROUND` constant defines the ground node to which capacitors are connected.
- The output is formatted in scientific notation with a precision of 1 before writing the RC lines and then reset to default.
- Node names are constructed by appending ":1" or ":2" to the `netName` for intermediate nodes.

```
// Example usage:
std::ofstream outFile("rc_lines.sp");
LibFile::generateRCLines(outFile, "net123", 5, false); // Generate intermediate RC lines for net
"net123", instance 5 LibFile::generateRCLines(outFile, "net123", 5, true);  // Generate final RC
lines for net "net123", instance 5 outFile.close();
```

Definition at line 931 of file LibFile.cpp.

Here is the caller graph for this function:

6.7.3.3 logic()

```
std::map< std::string, std::string > LibFile::logic (
                const std::string & cell_name )
```

Retrieves the logic functions for the output pins of a specified cell within the Liberty file.

This method searches for a cell with the given name in the parsed Liberty file (either by parsing the file directly or reading from a JSON representation). It then iterates through the output pins of the cell, extracting the logic function associated with each pin. The logic functions are stored in a map, where the key is the pin name and the value is the logic function string.

If the JSON file does not exist, the Liberty file is parsed, and a JSON file is created. If the JSON file exists but cannot be opened or parsed, an error is logged, and an empty map is returned. If the specified cell is not found or if no logic functions are found for the cell, a warning is logged.

Parameters

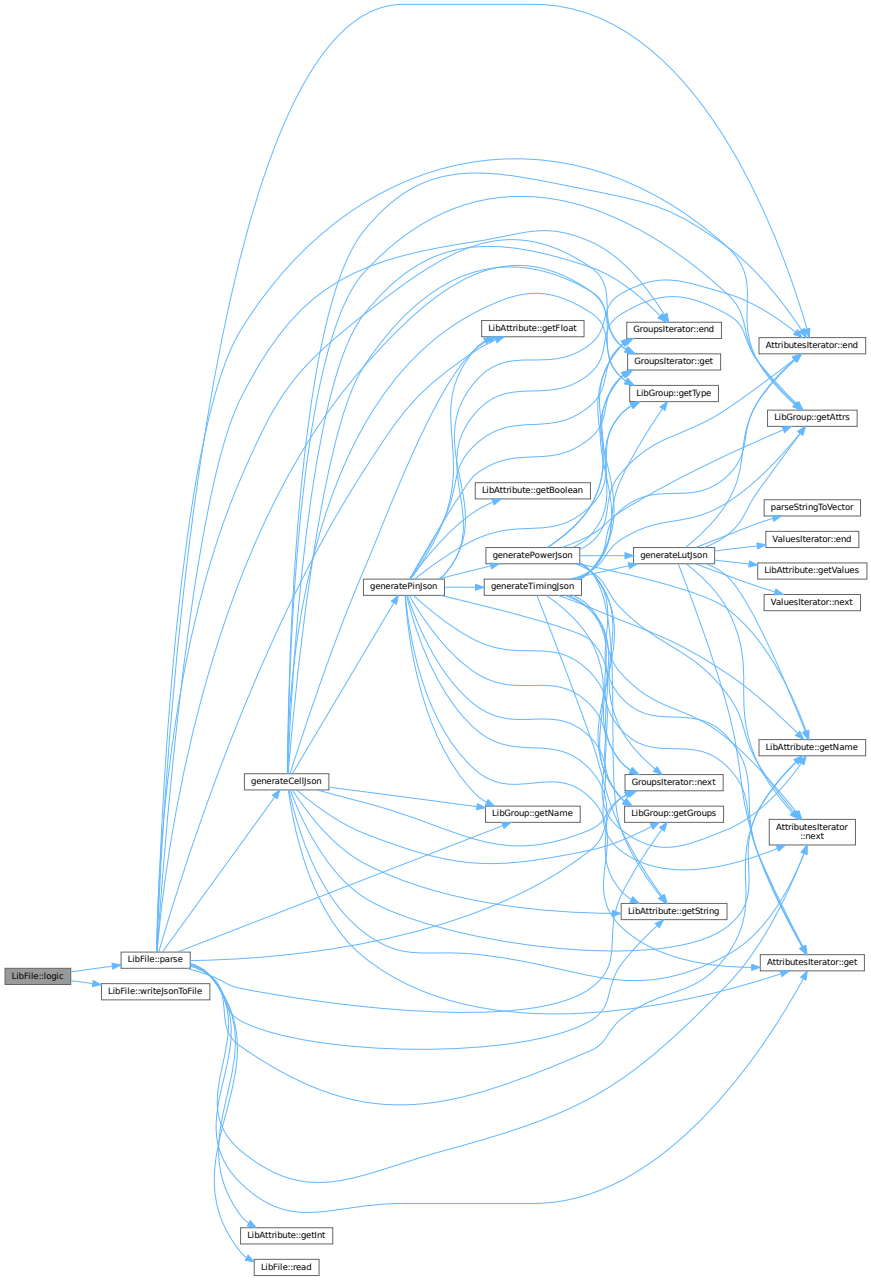
cell_name	The name of the cell for which to retrieve the logic functions.

Returns

A map containing the logic functions for the output pins of the specified cell. The keys of the map are the pin names, and the values are the corresponding logic function strings. Returns an empty map if no logic functions are found or if an error occurs during file processing.

Definition at line 1392 of file LibFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

6.7.3.4 modify()

```
void LibFile::modify ( )
```

Definition at line 182 of file LibFile.cpp.

6.7.3.5 modifySpiceNetlist()

```
bool LibFile::modifySpiceNetlist (
                const std::string & v2lvsSpiceFile,
                const std::string & finalSpiceFile,
                const std::string & targetGlobalLine )  [private]
```

Modifies a SPICE netlist generated by v2lvs, adding metadata, inserting a global line, and adjusting instance lines within subcircuits to include VDD/VSS connections and generate corresponding R/C lines.

This function reads a SPICE netlist from `v2lvsSpiceFile`, modifies it according to the following rules, and writes the result to `finalSpiceFile`:

1. **Metadata Comments:** Adds a header with application name, version, author, and timestamp.
2. **Global Line Insertion:** Inserts a specified global line (`targetGlobalLine`) after the first `.INCLUDE` directive.
3. **Subcircuit Modifications:**
   - Adds `VDD VSS` to the port list of `.SUBCKT` definitions.
   - For each instance line (starting with 'X' or 'x') within a subcircuit:
     - Modifies the output pin name by appending ":1".
     - Inserts `VDD VSS` before the module name in the instance line.
     - Calls `generateRCLines` to generate and insert R/C lines for the instance.
4. **Comment Handling:** Preserves original comments, but processes any pending instance line before writing the comment.
5. **Empty/v2lvs Header Line Skipping:** Skips empty lines and lines starting with v2lvs header comments.
6. ∗∗.GLOBAL Line Skipping:∗∗ Skips original .GLOBAL lines.

The function handles case-insensitive `.SUBCKT` and `.ENDS` directives. It also manages a `previous`↩ `InstanceLine` buffer to handle instance lines that might need modification before the next line is processed.

Parameters

v2lvsSpiceFile	The path to the input SPICE netlist file generated by v2lvs.
finalSpiceFile	The path to the output SPICE netlist file after modification. This file will be overwritten if it exists.
targetGlobalLine	The string containing the global line to be inserted after the `.INCLUDE` directive.

Returns

`true` if the modification was successful, `false` otherwise (e.g., if file opening fails).

Note

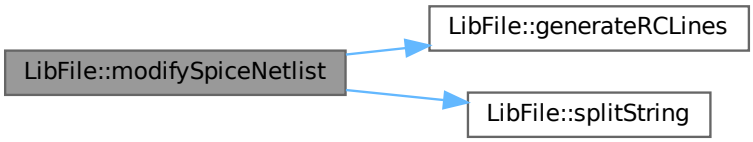
The function assumes that the second to last token in an instance line is the output net. It also assumes a specific format for instance lines where VDD/VSS insertion and R/C line generation are applicable. The `generateRCLines` method is expected to handle the actual generation of R/C components.

See also

generateRCLines

Definition at line 1013 of file LibFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

6.7.3.6 mono()

```
void LibFile::mono (
            const bool is_slew )
```

Validates that lookup tables are monotonically increasing with respect to the output load.

This function checks the following timing data in the current library to ensure monotonicity:

- cell_rise and retaining_rise
- cell_fall and retaining_fall
- rise_transition and retain_rise_slew
- fall_transition and retain_fall_slew
- min_pulse_width constraints (rise_constraint, fall_constraint)

The function first checks if a parsed JSON representation exists. If not, it parses the Liberty file and creates the JSON file. It then evaluates each timing arc in all cells and reports the pass/fail status at the end.

Parameters

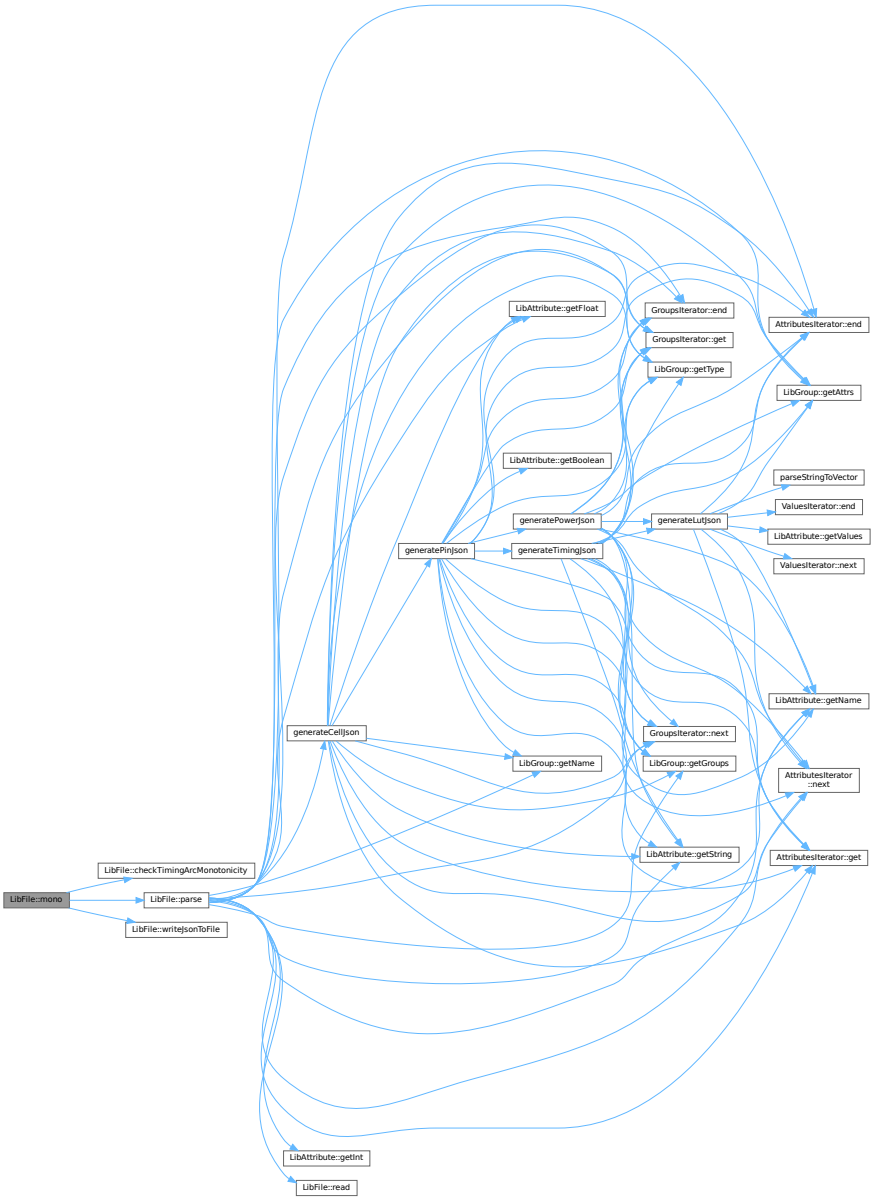
is_slew	Boolean flag indicating whether to check slew values rather than delay values

The function outputs:

- Number of cells that passed/failed the monotonicity check
- List of cells that failed the check (if any)

Definition at line 331 of file LibFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

6.7.3.7 parse()

```
void LibFile::parse ( )
```

Parses the Liberty file and extracts library information into a JSON structure.

This method handles the parsing of a Liberty file by:

1. Initializing the SI2 parser interface error handler
2. Reading the Liberty file contents
3. Extracting the library name
4. Processing second-level groups including:
    - Cell definitions, which are added to the JSON structure
    - Operating conditions, extracting PVT (Process, Voltage, Temperature) values

The parsed data is stored in the lib_json_ member variable, with the following structure:

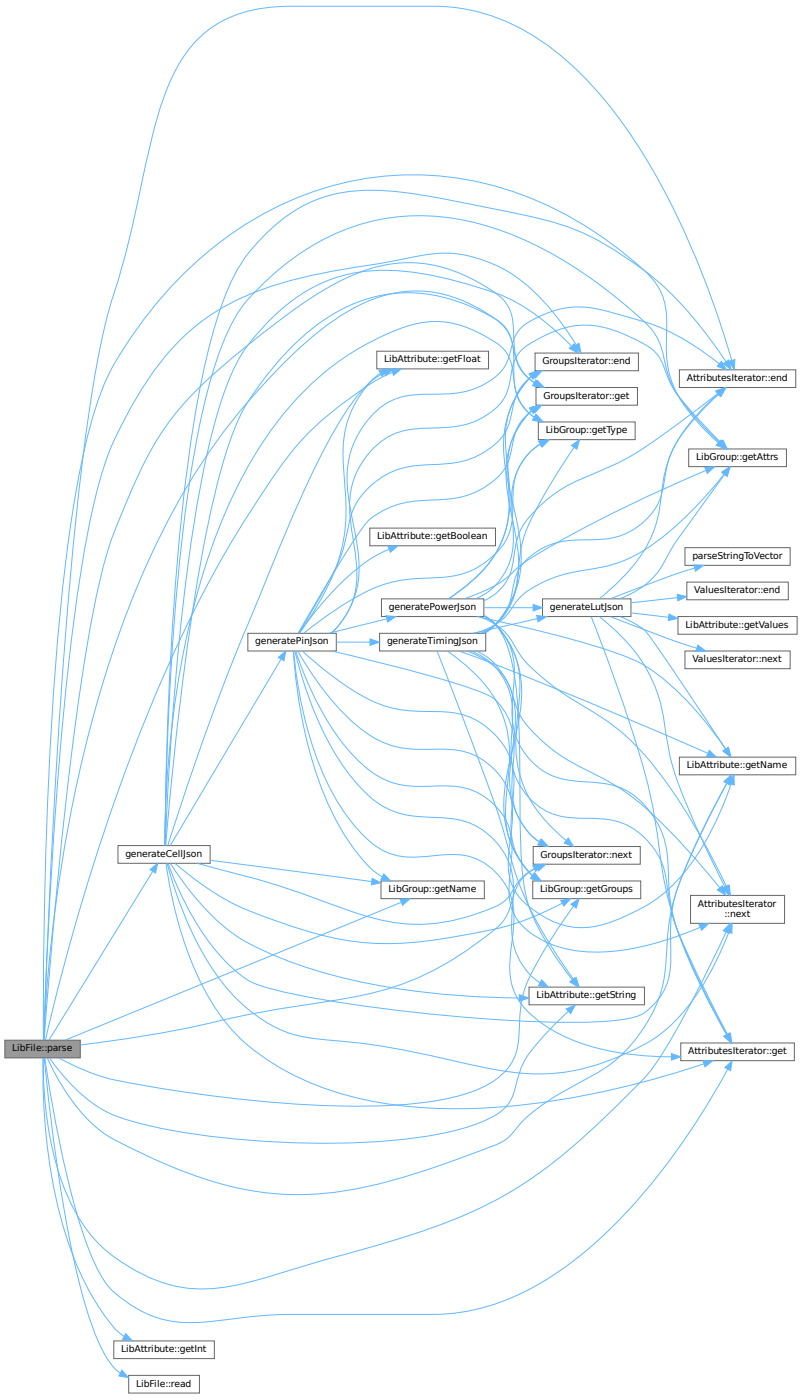
- "library_name": Name of the library
- "cells": Array of cell definitions
- "process": Process value
- "voltage": Voltage value (rounded to 2 decimal places)
- "temperature": Temperature value

Note

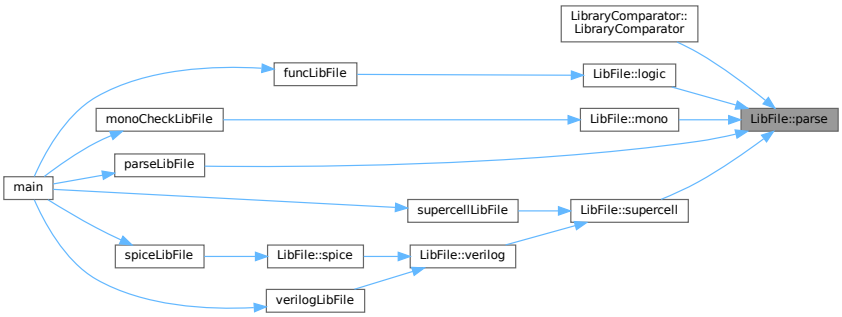
This method creates local scopes to ensure proper cleanup of SI2 Iterators.

Definition at line 116 of file LibFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



6.7.3.8 read()

```
void LibFile::read ( )  [private]
```

Reads the Liberty file specified by the filename_ member variable.

This function attempts to read a Liberty file and logs the process. It measures the time taken to read the file and logs any errors encountered during the read operation. If an error occurs, the function logs the error and terminates the program.

- Logs the start of the read operation.
- Measures the duration of the read operation.
- Checks for specific errors (invalid name or syntax error) and logs them.
- Terminates the program with specific exit codes if errors are detected.
- Logs the completion of the read operation and the time taken.

Note

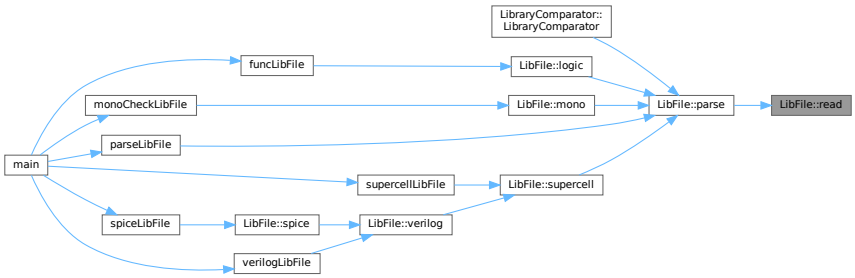
This function uses the spdlog library for logging and the si2drReadLibertyFile function for reading the Liberty file.

Exceptions

This	function will terminate the program with exit codes 301 or 401 if specific errors are encountered.

Definition at line 77 of file LibFile.cpp.

Here is the caller graph for this function:



6.7.3.9 spice()

```
void LibFile::spice (
                const int chain_length,
                const std::vector< std::string > & cell_names,
                const std::string & verilog_lib_file,
                const std::string & spice_lib_file )
```

Generates a SPICE netlist for the library based on a temporary Verilog file, using the V2LVS tool.

This function automates the process of converting a Verilog representation of the library into a SPICE netlist suitable for simulation. It involves the following steps:

1. Generates a temporary Verilog file using the `verilog` method.
2. Checks for the availability of the V2LVS tool in the system's PATH.
3. Constructs and executes the V2LVS command with appropriate options to generate an initial SPICE netlist.
4. Post-processes the generated SPICE netlist to adjust global signals using the `modifySpiceNetlist` method.
5. Logs the progress and any errors encountered during the process.

Parameters

chain_length	The length of the transistor chain for Verilog generation.
cell_names	A vector of cell names to be included in the Verilog generation.
verilog_lib_file	The path to the Verilog library file used by V2LVS for pin order information.
spice_lib_file	The path to the SPICE library file to be included in the generated SPICE netlist.

Note

The function assumes that the `verilog` method is available and correctly generates a temporary Verilog file.

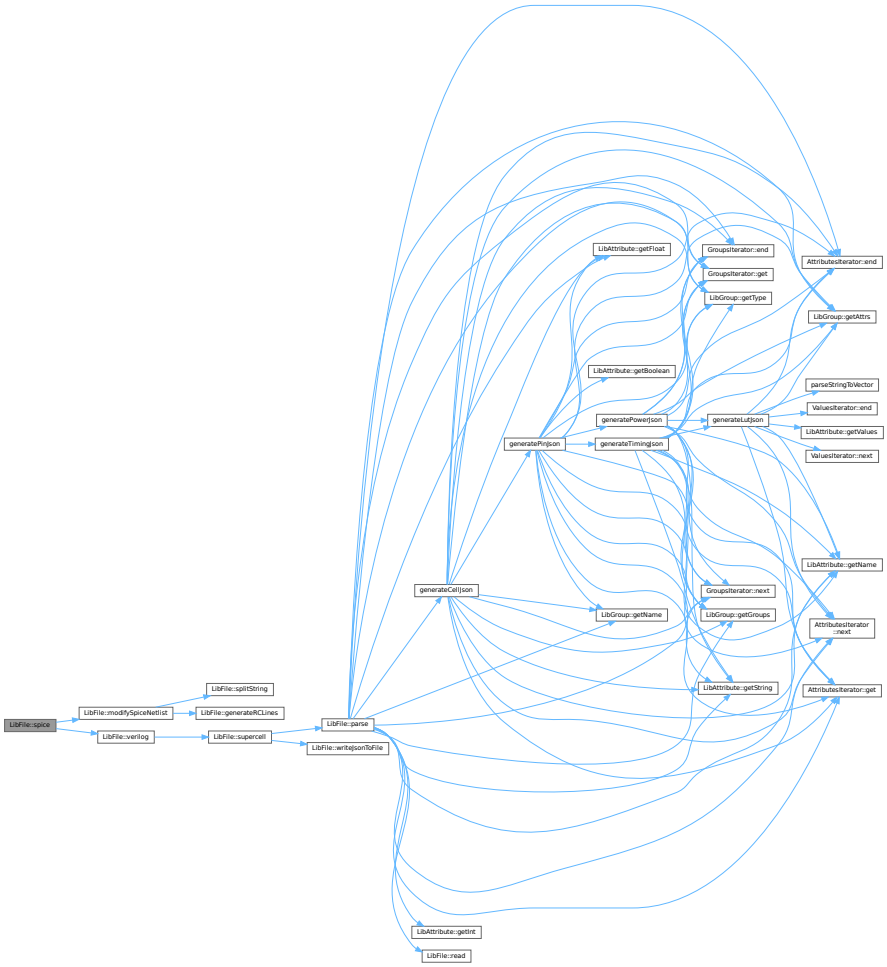
The V2LVS tool must be installed and accessible in the system's PATH for this function to work correctly.

The generated SPICE netlist is post-processed to adjust global signals.

Temporary files (Verilog and intermediate SPICE) are kept for debugging purposes.

Definition at line 1279 of file LibFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

6.7.3.10 splitString()

```
std::vector< std::string > LibFile::splitString (
            const std::string & s )  [private]
```

Splits a string into a vector of tokens, using whitespace as the delimiter.

This function takes a string as input and splits it into a vector of strings, where each element in the vector represents a token from the original string. Whitespace characters (spaces, tabs, newlines, etc.) are used as delimiters to separate the tokens.

Parameters

s	The string to be split.

Returns

A vector of strings, where each string is a token from the input string.

Definition at line 889 of file LibFile.cpp.

Here is the caller graph for this function:

6.7.3.11 supercell()

```
void LibFile::supercell (
                const int chain_length,
                const std::vector< std::string > & cell_names )
```

Creates supercells based on standard cells in the library file.

This method creates supercells by combining standard cells in chains. For each input-output pin pair of a cell, it creates a supercell entry with naming format: <cellname>**X**<**chain_length**><input_pin>↩ ___<output_pin>. The results are written to a .map file.

For sequential cells (with clock pins), the chain length is always set to 1 regardless of the requested chain length.

Parameters

chain_length	The length of the chain for combinational cells
cell_names	Vector of specific cell names to process. If empty, all cells will be processed.

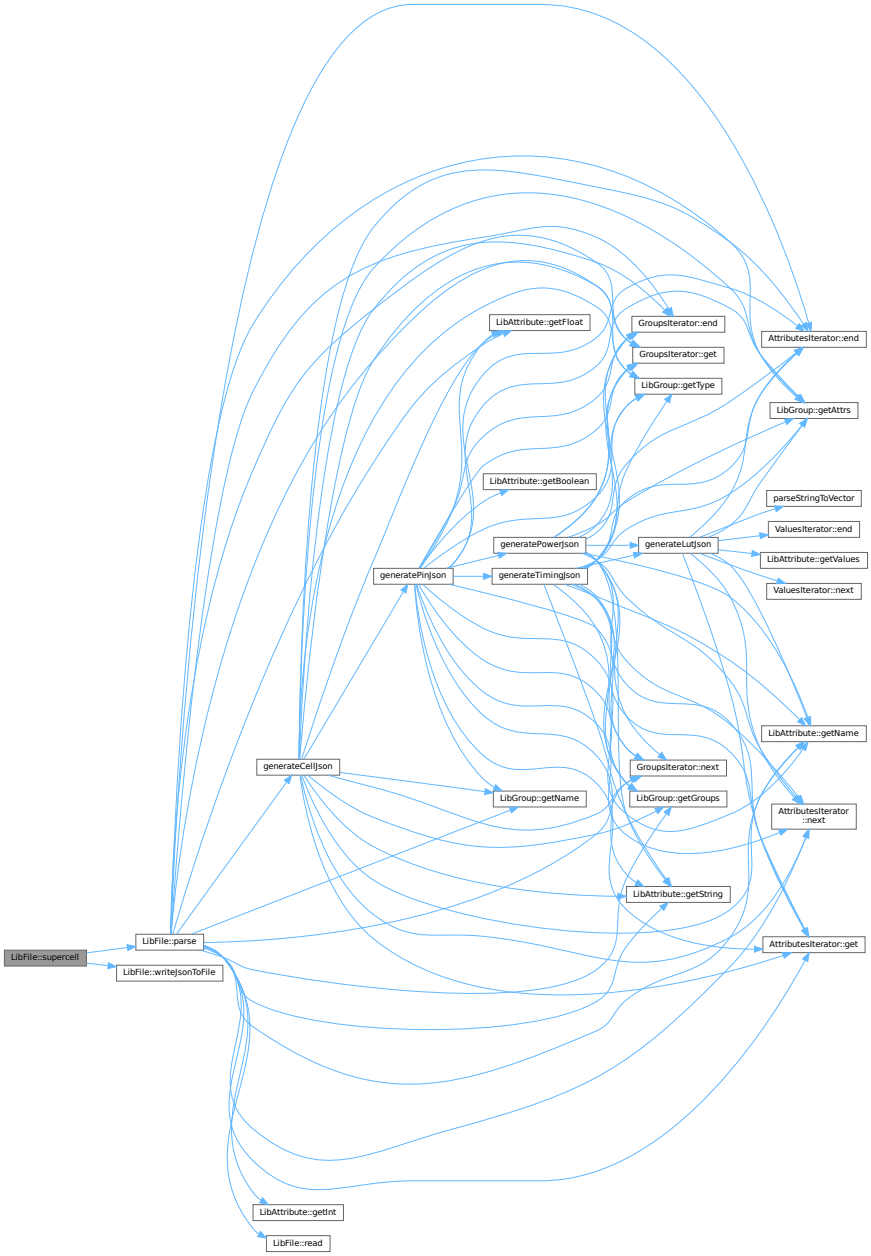
Note

Before creating supercells, this method checks for the existence of a JSON representation of the Liberty file and parses it if not found.

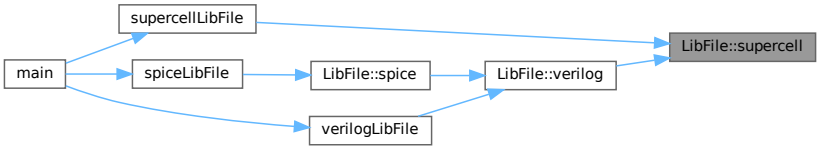
The method will report any requested cells that were not found in the library.

Definition at line 460 of file LibFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



6.7.3.12 verilog()

```
void LibFile::verilog (
                const int chain_length,
                const std::vector< std::string > & cell_names )
```

Generates Verilog files for cell validation based on the library file.

This function creates Verilog modules for each cell in the specified chain and generates a top-level module that connects them all together. The process involves:

1. Creating supercells based on the specified chain length and cell names
2. Reading the mapping file to identify supercell name relationships
3. Generating individual Verilog modules for each supercell, handling both sequential and combinational cells differently
4. Creating ANSI-style port definitions for each module
5. Using the slang library to build proper syntax trees for Verilog modules
6. Creating a top-level module that instantiates all the individual modules
7. Writing the complete Verilog output to the output file

Sequential cells are instantiated once, while combinational cells are instantiated multiple times based on the specified chain length.

Parameters

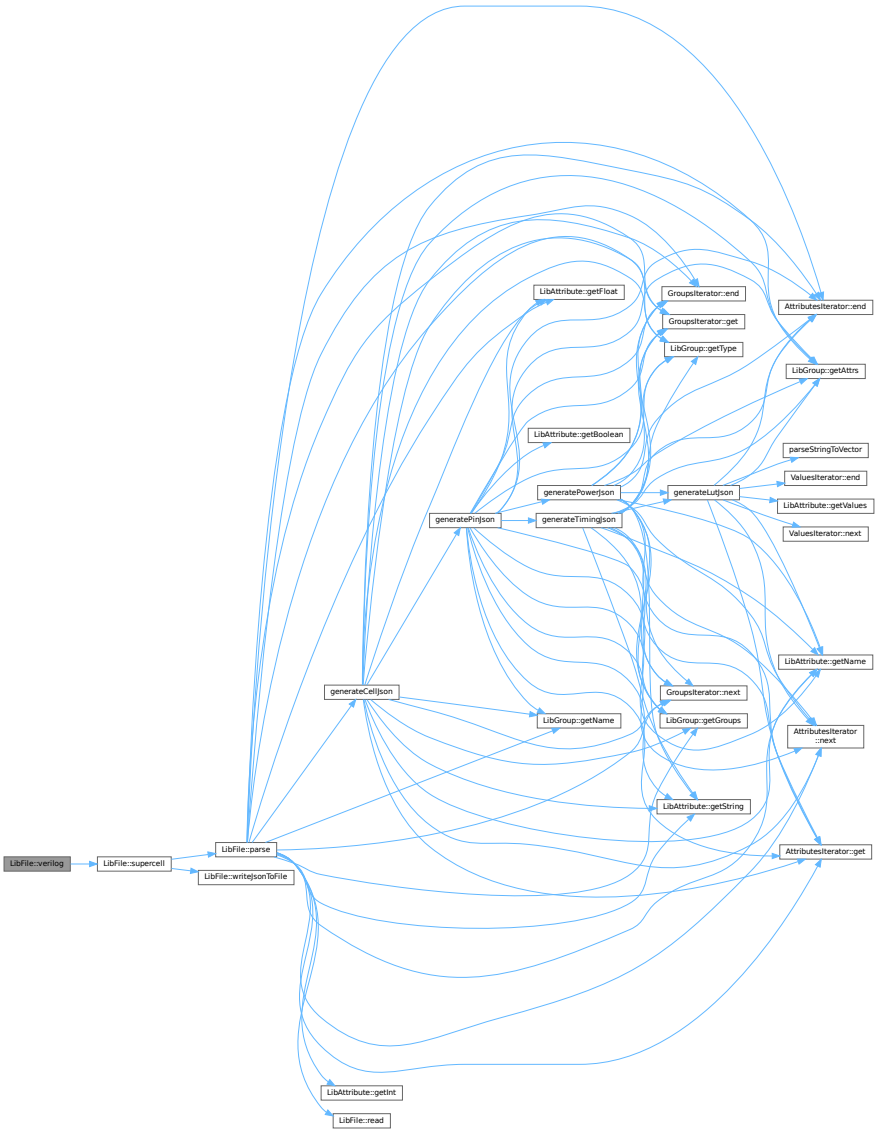
chain_length	The number of times to chain combinational cells together
cell_names	Vector of cell names to include in the Verilog generation

Note

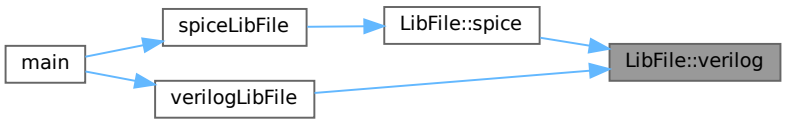
The function writes a temporary file during processing that might not be removed when the function completes (commented out cleanup code).

Definition at line 614 of file LibFile.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



6.7.3.13 writeJsonToFile()

```
void LibFile::writeJsonToFile ( )
```

Writes the JSON data stored in the object to a file.

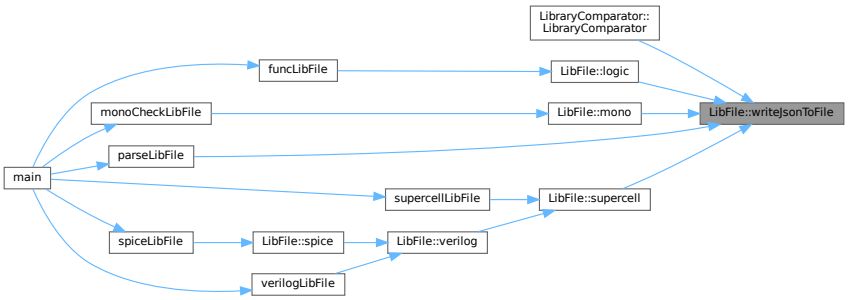
This method opens the file specified by jsonname_ and writes the content of lib_json_ with an indentation of 2 spaces. If the file cannot be opened, an error message is logged. Upon successful writing, an informational message is logged.

Exceptions

None	directly, but std::ofstream operations may throw exceptions

Definition at line 46 of file LibFile.cpp.

Here is the caller graph for this function:



6.7.4 Member Data Documentation

6.7.4.1 basename_

```
std::string LibFile::basename_
```

Definition at line 29 of file LibFile.hpp.

6.7.4.2 err_

```
si2drErrorT LibFile::err_  [private]
```

Definition at line 46 of file LibFile.hpp.

6.7.4.3 filename_

```
std::string LibFile::filename_
```

Definition at line 30 of file LibFile.hpp.

6.7.4.4 filepath_

```
std::filesystem::path LibFile::filepath_
```

Definition at line 28 of file LibFile.hpp.

6.7.4.5 jsonname_

```
std::string LibFile::jsonname_ = ""
```

Definition at line 32 of file LibFile.hpp.

6.7.4.6 lib_json_

```
json LibFile::lib_json_ = json::object()
```

Definition at line 34 of file LibFile.hpp.

6.7.4.7 libname_

```
std::string LibFile::libname_ = ""
```

Definition at line 31 of file LibFile.hpp.

6.7.4.8 logger_

```
std::shared_ptr<spdlog::logger> LibFile::logger_
```

Definition at line 27 of file LibFile.hpp.

6.7.4.9 loggername_

```
std::string LibFile::loggername_ = ""
```

Definition at line 33 of file LibFile.hpp.

6.7.4.10 process_

```
int LibFile::process_ = 0  [private]
```

Definition at line 47 of file LibFile.hpp.

6.7.4.11 temperature_

```
int LibFile::temperature_ = 0  [private]
```

Definition at line 49 of file LibFile.hpp.

6.7.4.12 voltage_

```
float LibFile::voltage_ = 0.0  [private]
```

Definition at line 48 of file LibFile.hpp.

The documentation for this class was generated from the following files:

- include/LibFile.hpp
- src/LibFile.cpp

6.8 LibGroup Class Reference

`#include <LibGroup.hpp>`

Public Member Functions

- LibGroup (si2drGroupIdT group, si2drErrorT &err)
- ~LibGroup ()
- std::string getName ()
- std::string getType ()
- si2drAttrsIdT getAttrs ()
- si2drGroupsIdT getGroups ()

Private Attributes

- si2drGroupIdT group_
- si2drErrorT & err_

6.8.1 Detailed Description

Definition at line 8 of file LibGroup.hpp.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 LibGroup()

```
LibGroup::LibGroup (
                si2drGroupIdT group,
                si2drErrorT & err )
```

Definition at line 3 of file LibGroup.cpp.

6.8.2.2 ~LibGroup()

```
LibGroup::~LibGroup ( )
```

Definition at line 5 of file LibGroup.cpp.

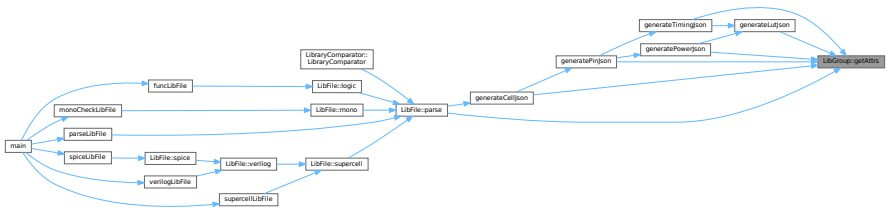
6.8.3 Member Function Documentation

6.8.3.1 getAttrs()

```
si2drAttrsIdT LibGroup::getAttrs ( )
```

Definition at line 19 of file LibGroup.cpp.

Here is the caller graph for this function:

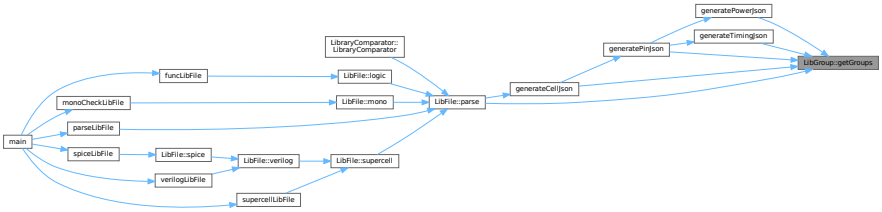


6.8.3.2 getGroups()

```
si2drGroupsIdT LibGroup::getGroups ( )
```

Definition at line 21 of file LibGroup.cpp.

Here is the caller graph for this function:

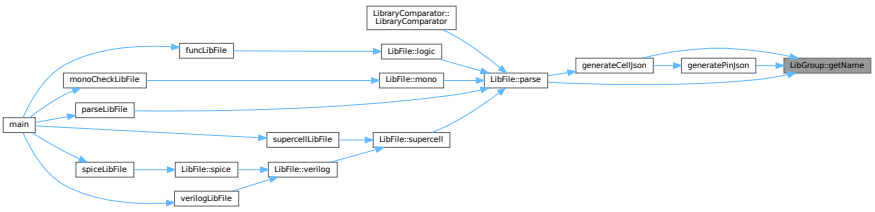


6.8.3.3 getName()

```
std::string LibGroup::getName ( )
```

Definition at line 7 of file LibGroup.cpp.

Here is the caller graph for this function:

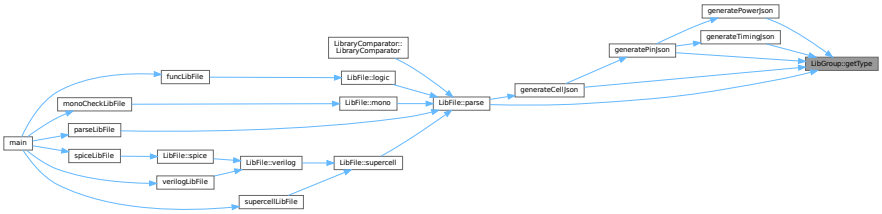


6.8.3.4 getType()

```
std::string LibGroup::getType ( )
```

Definition at line 14 of file LibGroup.cpp.

Here is the caller graph for this function:



6.8.4 Member Data Documentation

6.8.4.1 err_

```
si2drErrorT& LibGroup::err_  [private]
```

Definition at line 19 of file LibGroup.hpp.

6.8.4.2 group_

```
si2drGroupIdT LibGroup::group_  [private]
```

Definition at line 18 of file LibGroup.hpp.

The documentation for this class was generated from the following files:

- include/LibGroup.hpp
- src/LibGroup.cpp

6.9 LibraryComparator Class Reference

```
#include <LibraryComparator.hpp>
```

Public Member Functions

- LibraryComparator (LibFile &ref_libfile, LibFile &comp_libfile, double reltol, double abstol)

  Constructor for the LibraryComparator class.
- void generateReport (const std::string &output_file)

  Generates a comparison report between the reference and comparison libraries.

Public Attributes

- std::filesystem::path ref_lib_path_
- std::filesystem::path comp_lib_path_

Private Member Functions

- void compareCell (const std::string &cell_name, const json &ref_cell, const json &comp_cell, Table &table)

  Compares the output pins of a cell between a reference JSON and a comparison JSON.
- void comparePin (const std::string &cell_name, const std::string &pin_name, const json &ref_pin, const json &comp_pin, Table &table)

  Compares the timing arcs of a given pin between a reference JSON and a comparison JSON.
- void compareTimingArc (const std::string &cell_name, const std::string &pin_name, const std↩ ::string &timing_type, const json &ref_timing_arc, const json &comp_timing_arc, Table &table)

  Compares a timing arc between two JSON objects.
- void compareLut (const std::string &cell_name, const std::string &pin_name, const std::string &timing_type, const std::string &related_pin, const std::string &arc_name, const json &ref_lut, const json &comp_lut, Table &table)

  Compares lookup tables (LUTs) between reference and comparison libraries for a specific timing arc.

Private Attributes

- json ref_json_
- json comp_json_
- double reltol_
- double abstol_

6.9.1 Detailed Description

Definition at line 19 of file LibraryComparator.hpp.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 LibraryComparator()

```
LibraryComparator::LibraryComparator (
                LibFile & ref_libfile,
                LibFile & comp_libfile,
                double reltol,
                double abstol )
```

Constructor for the LibraryComparator class.

Initializes a LibraryComparator object by loading JSON data from reference and comparison library files. If JSON files don't exist, parses the library files first and creates the JSON files.

The constructor performs the following steps:

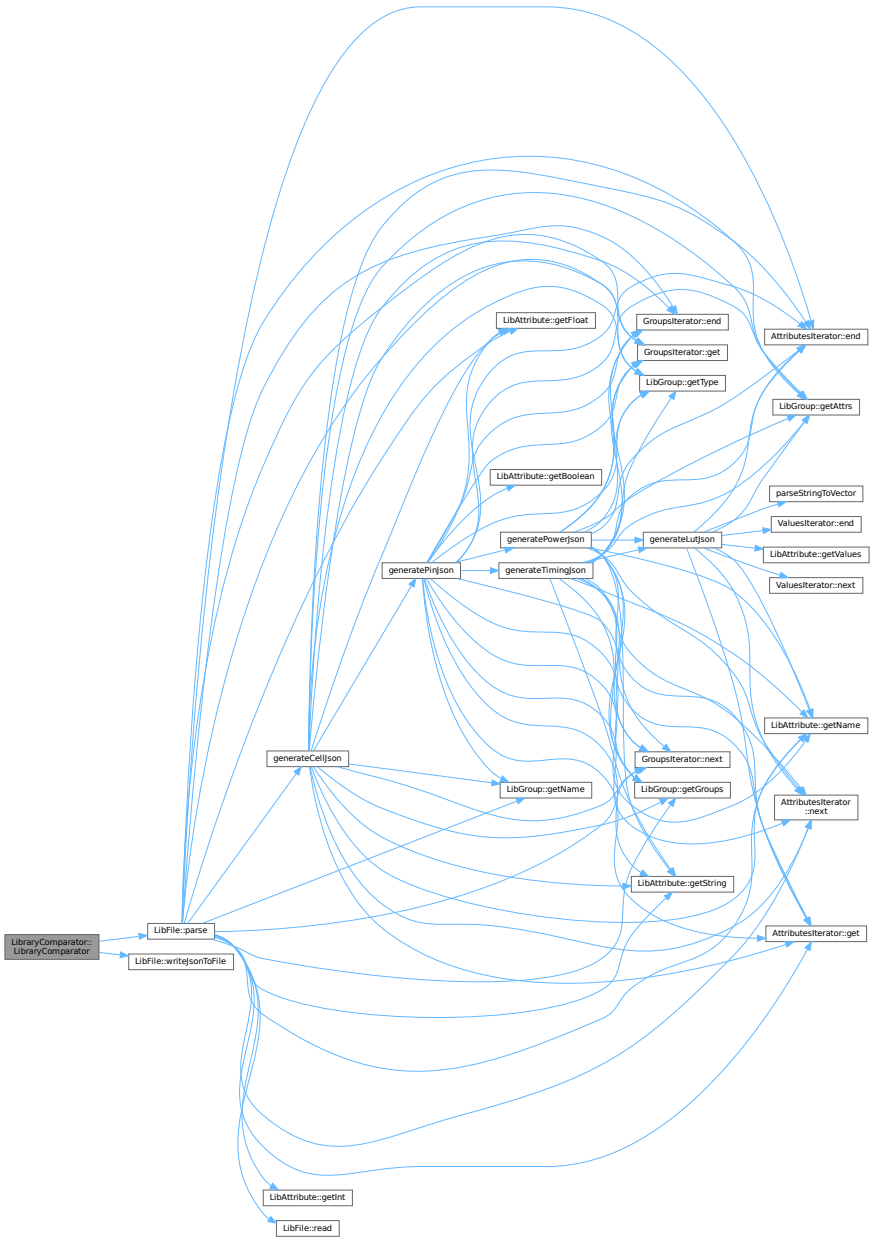
1. Looks for reference JSON file, parses library file if not found
2. Loads reference JSON data into ref_json_ member
3. Looks for comparison JSON file, parses library file if not found
4. Loads comparison JSON data into comp_json_ member
5. Stores file paths and initializes tolerance values for comparison

Parameters

ref_libfile	Reference library file object
comp_libfile	Comparison library file object
reltol	Relative tolerance for numerical comparisons (default defined elsewhere)
abstol	Absolute tolerance for numerical comparisons (default defined elsewhere)

Definition at line 21 of file LibraryComparator.cpp.

Here is the call graph for this function:



6.9.3 Member Function Documentation

6.9.3.1 compareCell()

```
void LibraryComparator::compareCell (
                const std::string & cell_name,
```

```
                const json & ref_cell,
                const json & comp_cell,
                Table & table )   [private]
```

Compares the output pins of a cell between a reference JSON and a comparison JSON.

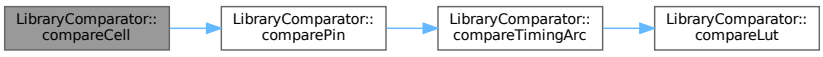
This function iterates through the output pins of the comparison cell and checks if each pin exists in the reference cell. If a pin is found in both cells, it calls the comparePin function to compare the details of the pin. If a pin is not found in the reference cell, a warning is logged. If the comparison cell does not contain any output pins, an informational message is logged.

Parameters

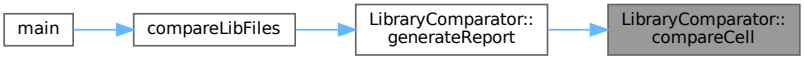
cell_name	The name of the cell being compared.
ref_cell	The reference JSON object containing the cell data.
comp_cell	The comparison JSON object containing the cell data.
table	The table object where the comparison results are stored.

Definition at line 301 of file LibraryComparator.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



6.9.3.2 compareLut()

```
void LibraryComparator::compareLut (
                const std::string & cell_name,
                const std::string & pin_name,
```

```
const std::string & timing_type,
const std::string & related_pin,
const std::string & arc_name,
const json & ref_lut,
const json & comp_lut,
Table & table )  [private]
```

Compares lookup tables (LUTs) between reference and comparison libraries for a specific timing arc.

This function compares the lookup table values between reference and comparison libraries for a given timing arc, checking the consistency of indices and values. It verifies that the indices match between libraries and then compares each value in the LUT against relative and absolute tolerance thresholds. Any discrepancies that exceed the tolerance limits are recorded in the provided table.

Parameters

cell_name	The name of the cell containing the LUT.
pin_name	The name of the pin associated with the LUT.
timing_type	The timing type of the arc (e.g., "rise_transition", "fall_transition").
related_pin	The related pin for the timing arc.
arc_name	The name of the specific timing arc being compared.
ref_lut	The lookup table from the reference library (in JSON format).
comp_lut	The lookup table from the comparison library (in JSON format).
table	Table object to store comparison results for any mismatches found.

Note

The function logs various information/warning/error messages:

- Logs index mismatches as warnings
- Logs value mismatches as debug messages
- Logs format errors in LUT structure as errors
- Records values exceeding tolerance in the provided table

The comparison uses both relative tolerance (reltol_) and absolute tolerance (abstol_) to determine if values differ significantly.

Definition at line 98 of file LibraryComparator.cpp.

Here is the caller graph for this function:

6.9.3.3 comparePin()

```
void LibraryComparator::comparePin (
                const std::string & cell_name,
                const std::string & pin_name,
                const json & ref_pin,
                const json & comp_pin,
                Table & table )  [private]
```

Compares the timing arcs of a given pin between a reference JSON and a comparison JSON.

This function iterates through the timing arcs of the comparison pin and looks for matching timing types in the reference pin. If a matching timing type is found, it calls the compareTimingArc function to compare the timing arcs. If a timing type is not found in the reference JSON, a warning is logged.

Parameters

cell_name	The name of the cell containing the pin.
pin_name	The name of the pin to compare.
ref_pin	The reference JSON object containing the pin's data.
comp_pin	The comparison JSON object containing the pin's data.
table	A reference to a Table object where comparison results are stored.

Definition at line 261 of file LibraryComparator.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

6.9.3.4 compareTimingArc()

```
void LibraryComparator::compareTimingArc (
                const std::string & cell_name,
                const std::string & pin_name,
                const std::string & timing_type,
                const json & ref_timing_arc,
                const json & comp_timing_arc,
                Table & table )  [private]
```

Compares a timing arc between two JSON objects.

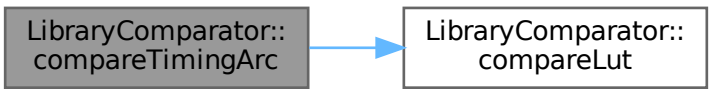
This function compares a specific timing arc (e.g., cell_rise, cell_fall, rise_transition, fall_transition) between a reference JSON object and a comparison JSON object. It extracts the related pin from the reference timing arc and iterates through a predefined list of arc names. If an arc name is found in the comparison timing arc, it checks if the same arc name exists in the reference timing arc. If both exist, it calls the compareLut function to compare the LUT data associated with the arc. If the arc name is found in the comparison JSON but not in the reference JSON, a warning message is logged.

Parameters

cell_name	The name of the cell.
pin_name	The name of the pin.
timing_type	The type of timing (e.g., "combinational", "sequential").
ref_timing_arc	The reference JSON object containing the timing arc information.
comp_timing_arc	The comparison JSON object containing the timing arc information.
table	The table to store comparison results.

Definition at line 220 of file LibraryComparator.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

6.9.3.5 generateReport()

```
void LibraryComparator::generateReport (
                const std::string & output_file )
```

Generates a comparison report between the reference and comparison libraries.

This function generates a detailed comparison report between the reference library and the comparison library. The report includes metadata such as the reference and comparison library paths, absolute and relative tolerances, and the application details. It also includes a legend explaining various symbols used in the report.

The function iterates through each cell in the comparison library, compares it with the corresponding cell in the reference library, and generates a table of differences. If there are any differences, it calculates summary statistics such as the average and maximum differences, and the number of outliers. The results are written to the specified output file in either Markdown or plain text format.

Parameters

output_file	The path to the output file where the report will be written.

Definition at line 343 of file LibraryComparator.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

6.9.4 Member Data Documentation

6.9.4.1 abstol_

```
double LibraryComparator::abstol_  [private]
```

Definition at line 30 of file LibraryComparator.hpp.

6.9.4.2 comp_json_

```
json LibraryComparator::comp_json_  [private]
```

Definition at line 28 of file LibraryComparator.hpp.

6.9.4.3 comp_lib_path_

```
std::filesystem::path LibraryComparator::comp_lib_path_
```

Definition at line 23 of file LibraryComparator.hpp.

6.9.4.4 ref_json_

```
json LibraryComparator::ref_json_  [private]
```

Definition at line 27 of file LibraryComparator.hpp.

6.9.4.5 ref_lib_path_

```
std::filesystem::path LibraryComparator::ref_lib_path_
```

Definition at line 22 of file LibraryComparator.hpp.

6.9.4.6 reltol_

```
double LibraryComparator::reltol_  [private]
```

Definition at line 29 of file LibraryComparator.hpp.

The documentation for this class was generated from the following files:

- include/LibraryComparator.hpp
- src/LibraryComparator.cpp

6.10 LogicComparator Class Reference

```
#include <LogicComparator.hpp>
```

Public Member Functions

- LogicComparator (const std::map< std::string, std::string > &ref_outpin_map, const std::map< std::string, std::string > &comp_outpin_map, const std::string &cell_name)
- void logic ()

  This function demonstrates the usage of the exprtk library to evaluate a boolean logic expression.
- std::string preprocessExpression (const std::string &input_expr)

  Preprocesses a logical expression string to prepare it for evaluation.
- bool extractVariables (const std::string &expr1, const std::string &expr2, std::vector< std::string > &sorted_vars)

  Extracts and validates variables from two expressions, ensuring they match.
- void compareSingleExpressionPair (const std::string &ref_expression_processed, const std::string &comp_expression_processed, const std::vector< std::string > &sorted_vars, PinComparisonResult &result)

  Compares two preprocessed logic expression strings for equivalence. Internal helper function.
- void compareCellLogic ()

  Compares logic for all output pins defined in the input maps, and stores results in all_pin_results_.
- void generateReport (const std::string &output_file)

  Generates a comparison report file based on cell logic comparison results.

Private Attributes

- std::map< std::string, std::string > ref_outpin_map_
- std::map< std::string, std::string > comp_outpin_map_
- std::string cell_name_
- std::map< std::string, PinComparisonResult > all_pin_results_

6.10.1 Detailed Description

Definition at line 38 of file LogicComparator.hpp.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 LogicComparator()

```
LogicComparator::LogicComparator (
                const std::map< std::string, std::string > & ref_outpin_map,
                const std::map< std::string, std::string > & comp_outpin_map,
                const std::string & cell_name )
```

Definition at line 3 of file LogicComparator.cpp.

6.10.3 Member Function Documentation

6.10.3.1 compareCellLogic()

```
void LogicComparator::compareCellLogic ( )
```

Compares logic for all output pins defined in the input maps, and stores results in all_pin_results_.

Compares the logic expressions for each output pin of a cell between a reference and a comparison design.

This method performs a detailed comparison of the logic expressions associated with each output pin of a cell. It iterates through each unique pin name found in both the reference and comparison designs, retrieves their corresponding logic expressions, preprocesses them, extracts and validates the variables used, and then compares the processed expressions. The results of each pin comparison, including any errors or discrepancies, are stored in the `all_pin_results_` map.

The comparison process includes the following steps:

1. **Pin Collection:** Collects all unique pin names from both the reference (`ref_outpin_map_`) and comparison (`comp_outpin_map_`) maps.
2. **Iteration:** Iterates through each unique pin name.
3. **Expression Retrieval:** Retrieves the raw logic expressions for the current pin from both the reference and comparison maps. If a pin is missing in either map, an error is logged, and the comparison is marked as impossible for that pin.
4. **Preprocessing:** Preprocesses the raw expressions to simplify them and remove any irrelevant characters or formatting. If preprocessing results in an empty expression, an error is logged, and the comparison is marked as impossible.
5. **Variable Extraction and Validation:** Extracts the variables used in both expressions and validates that the sets of variables match. If the variable sets do not match, an error is logged, and the comparison is marked as impossible.
6. **Expression Comparison:** Compares the preprocessed expressions using the extracted variables. This step determines whether the logic functions represented by the expressions are equivalent.
7. **Result Storage:** Stores the detailed results of the pin comparison, including the raw and processed expressions, any errors encountered, and the comparison result, in the `all_pin_results_` map.

Note

The `spdlog` library is used for logging information, warnings, and errors throughout the comparison process.

The `preprocessExpression` method is used to simplify the raw logic expressions before comparison.

The `extractVariables` method is used to extract and validate the variables used in the logic expressions.

The `compareSingleExpressionPair` method is used to compare the preprocessed expressions and determine whether they are equivalent.

See also

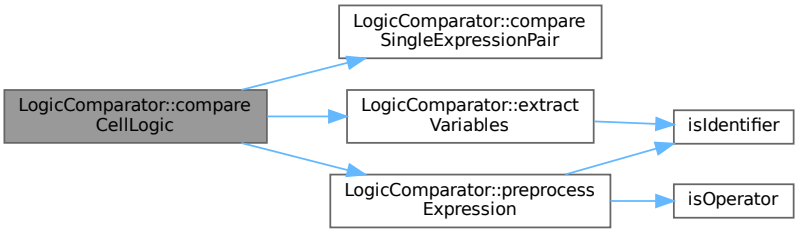
preprocessExpression

extractVariables

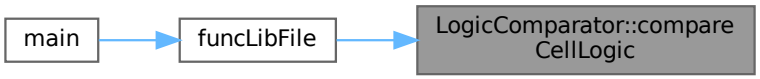
compareSingleExpressionPair

Definition at line 812 of file LogicComparator.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.2 compareSingleExpressionPair()

```
void LogicComparator::compareSingleExpressionPair (
                const std::string & ref_expression_processed,
                const std::string & comp_expression_processed,
                const std::vector< std::string > & sorted_vars,
                PinComparisonResult & result )
```

Compares two preprocessed logic expression strings for equivalence. Internal helper function.

Compares two boolean expressions for logical equivalence using a truth table.

Template Parameters

T	The numeric type used by ExprTk (e.g., double, float).

Parameters

ref_expression_processed	The preprocessed reference logic expression.

Parameters

comp_expression_processed	The preprocessed comparison logic expression.
sorted_vars	A sorted vector of unique variable names common to both expressions.
result	Reference to a PinComparisonResult object to store detailed results.

This function takes two processed boolean expressions (ref_expression_processed and comp_expression↩ _processed), a sorted list of variable names (sorted_vars), and a PinComparisonResult struct to store the results. It evaluates both expressions for all possible combinations of variable values and compares the resulting truth tables.

The function uses the ExprTk library to parse and evaluate the boolean expressions. It generates a truth table for each expression and compares the results. If the truth tables are identical, the expressions are considered logically equivalent.

Parameters

ref_expression_processed	The processed reference boolean expression.
comp_expression_processed	The processed comparison boolean expression.
sorted_vars	A sorted vector of variable names used in the expressions.
result	A PinComparisonResult struct to store the comparison results, including processed expressions, compilation status, equivalence, error messages, and the generated truth tables.

Note

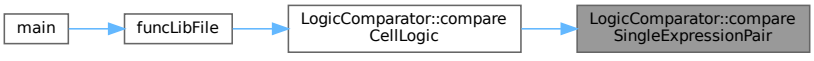
The function limits the number of variables to 20 to prevent excessively large truth tables. It also performs overflow checking on the number of combinations.

The function uses spdlog for logging debug, warning, and error messages.

The function assumes that the input expressions are valid boolean expressions containing only the variables listed in sorted_vars.

Definition at line 553 of file LogicComparator.cpp.

Here is the caller graph for this function:



6.10.3.3 extractVariables()

```
bool LogicComparator::extractVariables (
                const std::string & expr1_raw,
                const std::string & expr2_raw,
                std::vector< std::string > & sorted_vars )
```

Extracts and validates variables from two expressions, ensuring they match.

This function parses two raw expression strings (`expr1_raw` and `expr2_raw`) to identify potential variable names. It uses a regular expression to find identifiers and then validates them against a set of rules:

1. The identifier must be a valid identifier as determined by the `isIdentifier` function.
2. The identifier must not be a keyword or function name defined in the `keywords` set.

The function compares the sets of validated variables from both expressions. If the sets are identical, the variables are extracted, sorted alphabetically, and stored in the `sorted_vars` vector. If the sets differ, an error is reported, and the differences between the sets are logged.

Parameters

expr1_raw	The raw string representation of the first expression.
expr2_raw	The raw string representation of the second expression.
sorted_vars	A vector to store the sorted list of unique variable names if the variable sets from both expressions match. This vector is cleared if the sets do not match.

Returns

`true` if the variable sets from both expressions match, indicating that the `sorted_vars` vector contains the sorted list of unique variable names. `false` if the variable sets do not match, indicating an error.

Note

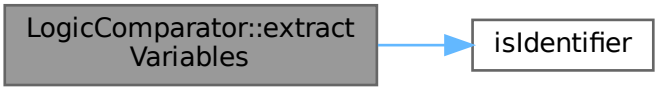
The `isIdentifier` function is used to validate potential variable names.

The `keywords` set contains a list of reserved words that cannot be used as variable names.

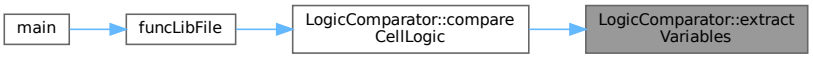
The function uses spdlog for logging debug, trace, info, and warning messages.

Definition at line 327 of file LogicComparator.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.3.4 generateReport()

```
void LogicComparator::generateReport (
                const std::string & output_file )
```

Generates a comparison report file based on cell logic comparison results.

Generates a comprehensive report comparing the logic equivalence of a cell's reference and comparison expressions.

Parameters

output_file	Path to the output report file.

This function performs the following steps:

1. **Initialization:** Sets up logging and determines the output format (Markdown or plain text) based on the file extension.
2. **File Handling:** Opens the specified output file in append mode, creating it if it doesn't exist. Handles potential file opening errors.

3. **Report Header:** Writes a header section to the report, including the cell name, application version, author, and timestamp.
4. **Legend Generation:** Creates a legend explaining the status symbols used in the report (e.g., [OK], [NO], [NA]).
5. **Table Generation:** Creates tables for reference pin functions, comparison pin functions, and the legend.
6. **Pin Result Processing:** Iterates through the results for each pin, generating detailed information including:
   - Pin Name
   - Reference and Comparison Truth Tables (if available)
   - Comparison Summary Table:
     - Status (Logic Equivalence)
     - Raw and Processed Expressions
     - Compilation Status
     - Error Messages (if any)
7. **Output:** Exports the generated tables and pin results to the output file in the determined format (Markdown or plain text).
8. **Closure:** Closes the output file and logs the completion of the report generation.

Parameters

output_file	The path to the output report file. If the file extension is ".md", the report will be generated in Markdown format; otherwise, it will be generated in plain text.

Note

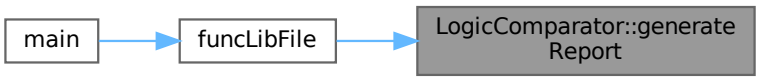
The function uses `spdlog` for logging and relies on several helper classes/structs:

- `Table`: For creating formatted tables.
- `MarkdownExporter`: For exporting tables to Markdown format.
- `LogicComparator::PinResult`: A struct containing the results of the logic comparison for a single pin.

The function appends to the output file if it already exists.

Definition at line 974 of file LogicComparator.cpp.

Here is the caller graph for this function:



6.10.3.5 logic()

```
void LogicComparator::logic ( )
```

This function demonstrates the usage of the exprtk library to evaluate a boolean logic expression.

It defines a boolean expression "not(A and B) or C" and evaluates it for all possible combinations of boolean values for the variables A, B, and C. The results are then printed to the console in a tabular format.

The function utilizes the exprtk library for:

- Defining a symbol table to hold the variables A, B, and C.
- Creating an expression object and registering the symbol table with it.
- Compiling the boolean expression string into the expression object.
- Iterating through all possible boolean combinations for A, B, and C.
- Assigning the boolean values to the variables in the symbol table.
- Evaluating the expression using expression.value().
- Printing the input values and the result to the console.

Definition at line 24 of file LogicComparator.cpp.

Here is the caller graph for this function:

6.10.3.6 preprocessExpression()

```
std::string LogicComparator::preprocessExpression (
                const std::string & input_expr )
```

Preprocesses a logical expression string to prepare it for evaluation.

This function performs several steps to transform the input expression:

1. Trims leading/trailing whitespace.
2. Replaces the logical NOT operator '!' with " not " (except for '!=').
3. Adds spaces around operators (+, ^, *) and parentheses.
4. Replaces symbolic operators (+, ^, *) with their keyword equivalents (or, xor, and).
5. Tokenizes the expression based on spaces.
6. Inserts implied 'and' operators between operands where necessary. For example, "A B" becomes "A and B".
7. Handles 'not Identifier' sequences by converting them to 'not(Identifier)'.
8. Reconstructs the final expression string from the processed tokens, adding spaces appropriately.
9. Performs a final cleanup to consolidate multiple spaces and trim the result.

Parameters

input_expr	The input logical expression string.

Returns

The preprocessed logical expression string, ready for evaluation. Returns an empty string if the input is empty or if no tokens are found after processing.

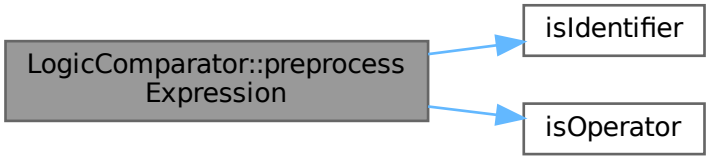
Note

The function uses spdlog for debugging and tracing.

The function assumes the existence of helper functions `isIdentifier` and `isOperator` to classify tokens.

Definition at line 128 of file LogicComparator.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

6.10.4 Member Data Documentation

6.10.4.1 all_pin_results_

```
std::map<std::string, PinComparisonResult> LogicComparator::all_pin_results_  [private]
```

Definition at line 86 of file LogicComparator.hpp.

6.10.4.2 cell_name_

```
std::string LogicComparator::cell_name_  [private]
```

Definition at line 85 of file LogicComparator.hpp.

6.10.4.3 comp_outpin_map_

```
std::map<std::string, std::string> LogicComparator::comp_outpin_map_  [private]
```

Definition at line 84 of file LogicComparator.hpp.

6.10.4.4 ref_outpin_map_

```
std::map<std::string, std::string> LogicComparator::ref_outpin_map_  [private]
```

Definition at line 83 of file LogicComparator.hpp.

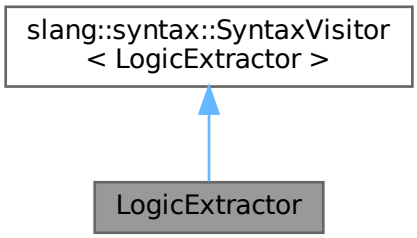
The documentation for this class was generated from the following files:

- include/LogicComparator.hpp
- src/LogicComparator.cpp

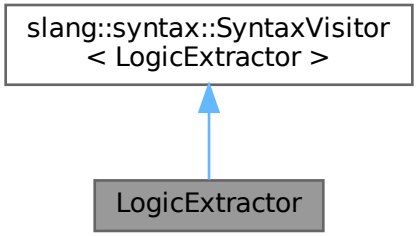
6.11 LogicExtractor Class Reference

```
#include <LogicExtractor.hpp>
```

Inheritance diagram for LogicExtractor:



Collaboration diagram for LogicExtractor:



Public Member Functions

- LogicExtractor (const std::string &targetCell)
- void handle (const slang::syntax::ModuleDeclarationSyntax &module)

  Handles a module declaration syntax node.
- void handle (const slang::syntax::PortDeclarationSyntax &portDecl)

  Handles a port declaration syntax node.
- void handle (const slang::syntax::NonAnsiPortListSyntax &portList)

  Handles a non-ANSI port list in a module.
- void handle (const slang::syntax::NetDeclarationSyntax &netDecl)

  Handles a net declaration syntax node.
- void handle (const slang::syntax::PrimitiveInstantiationSyntax &primitiveInst)

  Handles the extraction of logic gate information from a primitive instantiation syntax node.
- const std::unordered_map< std::string, GateInfo > & getExtractedGates () const
- const std::unordered_set< std::string > & getPrimaryInputs () const
- const std::unordered_set< std::string > & getPrimaryOutputs () const
- const std::unordered_set< std::string > & getInternalWires () const
- std::map< std::string, std::string > getLogicExpressions ()

  Extracts logic expressions for all primary output ports of the target cell.

Private Member Functions

- std::string deriveLogicRecursive (const std::string &signalName)

  Recursively derives the logic expression for a given signal.
- std::string formatExpression (const GateInfo &gateInfo, const std::vector< std::string > &input↩ Exprs)

  Formats a logic expression based on the gate type and input expressions.

Private Attributes

- const std::string & targetCell_
- bool inTargetModule_
- bool parsingComplete_
- std::unordered_set< std::string > primaryInputs_
- std::unordered_set< std::string > primaryOutputs_
- std::unordered_set< std::string > internalWires_
- std::unordered_map< std::string, std::string > portDirections_
- std::unordered_map< std::string, GateInfo > gateOutputDrivers_
- std::unordered_map< std::string, std::string > logicCache_

6.11.1 Detailed Description

Definition at line 17 of file LogicExtractor.hpp.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 LogicExtractor()

```
LogicExtractor::LogicExtractor (
                const std::string & targetCell )  [inline], [explicit]
```

Definition at line 20 of file LogicExtractor.hpp.

6.11.3 Member Function Documentation

6.11.3.1 deriveLogicRecursive()

```
std::string LogicExtractor::deriveLogicRecursive (
                const std::string & signalName )  [private]
```

Recursively derives the logic expression for a given signal.

This function traces back the signal to its driving gates and primary inputs, constructing a logic expression that represents the signal's behavior. It uses memoization to cache previously computed expressions, avoiding redundant calculations.

Parameters

signalName	The name of the signal for which to derive the logic expression.

Returns

A string representing the logic expression for the signal.

Exceptions

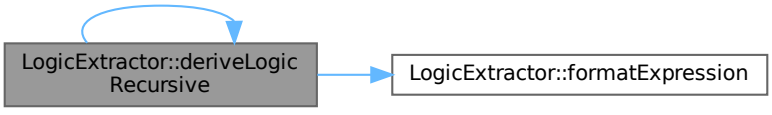
std::runtime_error	if the signal is not a primary input, a known wire, or driven by a recognized gate/assignment. Also thrown if an empty input signal name is encountered or if an internal wire has no identified driver.

The function operates in the following steps:

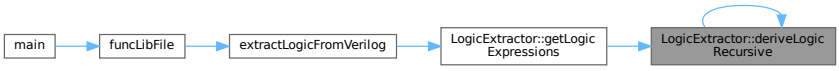
1. **Check Cache (Memoization):** If the logic expression for the signal is already cached in `logic↩ Cache_`, it is immediately returned.
2. **Base Case: Is it a primary input?** If the signal is a primary input (present in `primaryInputs_`), its logic expression is simply its own name.
3. **Recursive Step: Is it driven by a gate?** If the signal is driven by a gate (present in `gate↩ OutputDrivers_`), the function recursively derives the logic expressions for all input signals of the gate. These input expressions are then used to format the overall expression for the current signal based on the gate type.
4. **Handle Assign statements:** (Currently not implemented but reserved for future use).
5. **Error Case:** If the signal is not found in any of the above categories, it indicates an error. An exception is thrown, indicating that the signal is either an undriven internal wire or an unknown signal.

Definition at line 510 of file LogicExtractor.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



6.11.3.2 formatExpression()

```
std::string LogicExtractor::formatExpression (
                const GateInfo & gateInfo,
                const std::vector< std::string > & inputExprs )  [private]
```

Formats a logic expression based on the gate type and input expressions.

This function takes a GateInfo structure describing the gate and a vector of input expressions (strings) and constructs a logic expression string representing the gate's operation. It handles AND, NAND, OR, NOR, XOR, XNOR, NOT, and BUF gates. It uses '∗', '+', '^', and '!' operators for AND, OR, XOR, and NOT respectively. If the gate type is unsupported, or if the number of inputs is incorrect for NOT/BUF gates, or if a gate requiring inputs receives none, an error string is returned, and a warning is logged.

Parameters

gateInfo	A GateInfo struct containing information about the gate, including its type (slang::parsing::TokenKind) and name.
inputExprs	A vector of strings representing the input expressions to the gate.

Returns

A string representing the formatted logic expression, or an error string if formatting fails due to unsupported gate type or incorrect number of inputs.

Definition at line 584 of file LogicExtractor.cpp.

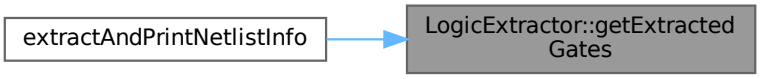
Here is the caller graph for this function:

6.11.3.3 getExtractedGates()

```
const std::unordered_map< std::string, GateInfo > & LogicExtractor::getExtractedGates ( ) const  [inline]
```

Definition at line 37 of file LogicExtractor.hpp.

Here is the caller graph for this function:

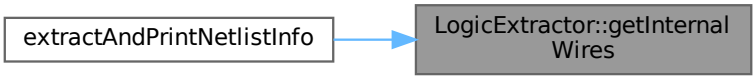


6.11.3.4 getInternalWires()

```
const std::unordered_set< std::string > & LogicExtractor::getInternalWires ( ) const  [inline]
```

Definition at line 42 of file LogicExtractor.hpp.

Here is the caller graph for this function:



6.11.3.5 getLogicExpressions()

```
std::map< std::string, std::string > LogicExtractor::getLogicExpressions ( )
```

Extracts logic expressions for all primary output ports of the target cell.

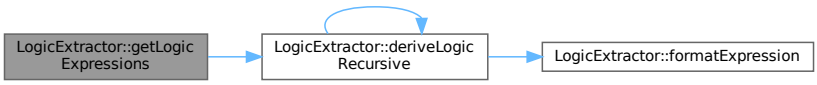
This method iterates through the primary output ports, derives the logic expression for each, and stores the result in a map. If an error occurs during the derivation of logic for a particular output, an error message is stored in the map instead.

Returns

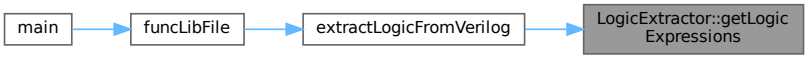
A map where the key is the output port name (std::string) and the value is the corresponding logic expression (std::string). If AST parsing is not complete or the target module is not found, an empty map is returned. If an error occurs during logic derivation for a specific output, the corresponding value in the map will be an error message.

Definition at line 451 of file LogicExtractor.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

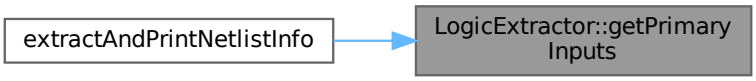


6.11.3.6 getPrimaryInputs()

```
const std::unordered_set< std::string > & LogicExtractor::getPrimaryInputs ( ) const  [inline]
```

Definition at line 40 of file LogicExtractor.hpp.

Here is the caller graph for this function:

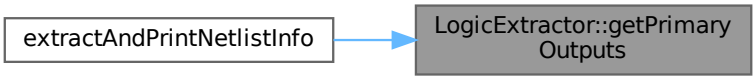


6.11.3.7 getPrimaryOutputs()

```
const std::unordered_set< std::string > & LogicExtractor::getPrimaryOutputs ( ) const  [inline]
```

Definition at line 41 of file LogicExtractor.hpp.

Here is the caller graph for this function:



6.11.3.8 handle() [1/5]

```
void LogicExtractor::handle (
                const slang::syntax::ModuleDeclarationSyntax & module )
```

Handles a module declaration syntax node.

This function is the entry point for processing a module declaration. It checks if the module is the target module and, if so, extracts relevant information such as primary inputs, primary outputs, and internal wires. It also visits the children of the module node to process ports, declarations, and instances.

The function ensures that the target module is only processed once and that parsing stops after the target module is found.

Parameters

module	A reference to the module declaration syntax node.

Definition at line 18 of file LogicExtractor.cpp.

6.11.3.9 handle() [2/5]

```
void LogicExtractor::handle (
                const slang::syntax::NetDeclarationSyntax & netDecl )
```

Handles a net declaration syntax node.

This function processes a net declaration syntax node, extracting the names of declared wires. It checks if the current scope is within the target module and avoids adding ports again if they were already declared as nets. The extracted wire names are added to the internal wires set.

Parameters

netDecl	A reference to the NetDeclarationSyntax node to handle.

Definition at line 277 of file LogicExtractor.cpp.

6.11.3.10 handle() [3/5]

```
void LogicExtractor::handle (
                const slang::syntax::NonAnsiPortListSyntax & portList )
```

Handles a non-ANSI port list in a module.

This method iterates through the ports in a non-ANSI port list, extracts the port name, determines its direction (input, output, or inout), and adds it to the appropriate sets (primaryInputs_, primaryOutputs_, internalWires_). It also handles different types of port declarations, including implicit, explicit, and empty ports.

Parameters

portList	A reference to the NonAnsiPortListSyntax object representing the port list.

- Skips processing if not in the target module (inTargetModule_ is false).
- Extracts port names from ImplicitNonAnsiPortSyntax and ExplicitNonAnsiPortSyntax.
- Handles PortConcatenationSyntax by logging a warning and skipping the port.
- Skips EmptyNonAnsiPortSyntax (placeholders).
- Uses the portDirections_ map to determine the direction of each port.
- Adds ports to primaryInputs_ and internalWires_ if the direction is "input".
- Adds ports to primaryOutputs_ and internalWires_ if the direction is "output".
- Adds ports to both primaryInputs_ and primaryOutputs_ if the direction is "inout".
- Logs warnings for ports with unknown or missing directions.

- Logs warnings if the port name cannot be determined.

Note

This method does not call visitDefault.

Definition at line 179 of file LogicExtractor.cpp.

6.11.3.11 handle() [4/5]

```
void LogicExtractor::handle (
                const slang::syntax::PortDeclarationSyntax & portDecl )
```

Handles a port declaration syntax node.

This method extracts information about port declarations within a SystemVerilog module, including the port's direction (input, output, or inout) and name. It updates internal data structures to track primary inputs, primary outputs, internal wires, and port directions.

Parameters

portDecl	A reference to the PortDeclarationSyntax node being processed.

- It first checks if the current processing context is within the target module. If not, the method returns early.
- It determines the port's direction by inspecting the port header (VariablePortHeaderSyntax or Net↩ PortHeaderSyntax). The direction is extracted using `valueText()` to handle keywords represented as text tokens.
- It iterates through the declarators in the port declaration to extract port names.
- If multiple direction declarations are found for the same port, a warning is logged, and the first declared direction is retained.
- Based on the port's direction, the port name is added to the appropriate sets:
  - `primaryInputs_`: For input and inout ports.
  - `primaryOutputs_`: For output and inout ports.
  - `internalWires_`: For all ports (input, output, inout, and ports with unknown directions).
- If a port has an unknown or missing direction, it's treated as an internal wire, and a warning is logged.

- The method avoids calling `visitDefault` to prevent unexpected revisits of syntax nodes.

Note

The `logicCache_` update is commented out, indicating it's part of a later processing step.

Warning

Logs warnings for port declarations without headers, port declarators without names, and multiple direction declarations for the same port.

Definition at line 92 of file LogicExtractor.cpp.

6.11.3.12 handle() [5/5]

```
void LogicExtractor::handle (
            const slang::syntax::PrimitiveInstantiationSyntax & primitiveInst )
```

Handles the extraction of logic gate information from a primitive instantiation syntax node.

This method processes a primitive instantiation, extracting the gate type, input signals, and output signal. It populates the `gateOutputDrivers_` map, which stores the driving gate information for each output signal. It also identifies internal wires and checks for multiple drivers on the same signal.

Parameters

primitiveInst	A reference to the PrimitiveInstantiationSyntax node representing the gate instance.

Definition at line 309 of file LogicExtractor.cpp.

6.11.4 Member Data Documentation

6.11.4.1 gateOutputDrivers_

```
std::unordered_map<std::string, GateInfo> LogicExtractor::gateOutputDrivers_  [private]
```

Definition at line 63 of file LogicExtractor.hpp.

6.11.4.2 inTargetModule_

```
bool LogicExtractor::inTargetModule_  [private]
```

Definition at line 50 of file LogicExtractor.hpp.

6.11.4.3 internalWires_

```
std::unordered_set<std::string> LogicExtractor::internalWires_  [private]
```

Definition at line 56 of file LogicExtractor.hpp.

6.11.4.4 logicCache_

```
std::unordered_map<std::string, std::string> LogicExtractor::logicCache_  [private]
```

Definition at line 66 of file LogicExtractor.hpp.

6.11.4.5 parsingComplete_

```
bool LogicExtractor::parsingComplete_  [private]
```

Definition at line 51 of file LogicExtractor.hpp.

6.11.4.6 portDirections_

```
std::unordered_map<std::string, std::string> LogicExtractor::portDirections_  [private]
```

Definition at line 60 of file LogicExtractor.hpp.

6.11.4.7 primaryInputs_

```
std::unordered_set<std::string> LogicExtractor::primaryInputs_  [private]
```

Definition at line 54 of file LogicExtractor.hpp.

6.11.4.8 primaryOutputs_

```
std::unordered_set<std::string> LogicExtractor::primaryOutputs_  [private]
```

Definition at line 55 of file LogicExtractor.hpp.

6.11.4.9 targetCell_

```
const std::string& LogicExtractor::targetCell_  [private]
```

Definition at line 49 of file LogicExtractor.hpp.

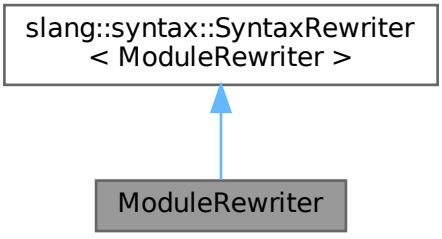
The documentation for this class was generated from the following files:

- include/LogicExtractor.hpp
- src/LogicExtractor.cpp

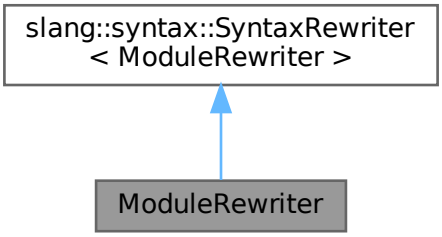
6.12 ModuleRewriter Class Reference

`#include <verilog_utils.hpp>`

Inheritance diagram for ModuleRewriter:



Collaboration diagram for ModuleRewriter:



Public Member Functions

- ModuleRewriter (const std::vector< std::string > &inputPins, const std::vector< std::string > &outputPins, const std::pair< std::string, std::string > &supercell_entry, int instance_count, std::shared_ptr< spdlog::logger > logger)
- void handle (const slang::syntax::SyntaxNode &node)

  Processes a syntax node in the module's AST.
- void handle (const slang::syntax::ModuleDeclarationSyntax &module)

Public Attributes

- std::shared_ptr< spdlog::logger > logger_

Private Attributes

- const std::vector< std::string > & inputPins_
- const std::vector< std::string > & outputPins_
- const std::string cellName_
- const std::string moduleName_
- std::map< std::string, std::string > portInfoMap_
- int depth_
- int instance_count_

6.12.1 Detailed Description

Definition at line 59 of file verilog_utils.hpp.

6.12.2 Constructor & Destructor Documentation

6.12.2.1 ModuleRewriter()

```
ModuleRewriter::ModuleRewriter (
                const std::vector< std::string > & inputPins,
                const std::vector< std::string > & outputPins,
                const std::pair< std::string, std::string > & supercell_entry,
                int instance_count,
                std::shared_ptr< spdlog::logger > logger )  [inline], [explicit]
```

Definition at line 61 of file verilog_utils.hpp.

6.12.3 Member Function Documentation

6.12.3.1 handle() [1/2]

```
void ModuleRewriter::handle (
                const slang::syntax::ModuleDeclarationSyntax & module )
```

Definition at line 406 of file verilog_utils.cpp.

6.12.3.2 handle() [2/2]

```
void ModuleRewriter::handle (
                const slang::syntax::SyntaxNode & node )
```

Processes a syntax node in the module's AST.

This method is called for each syntax node during the traversal of the AST. It logs debug information about the current node and continues processing its child nodes by recursively calling the appropriate visit methods.

Parameters

node	The syntax node to process

Definition at line 396 of file verilog_utils.cpp.

6.12.4 Member Data Documentation

6.12.4.1 cellName_

```
const std::string ModuleRewriter::cellName_  [private]
```

Definition at line 75 of file verilog_utils.hpp.

6.12.4.2 depth_

```
int ModuleRewriter::depth_  [private]
```

Definition at line 78 of file verilog_utils.hpp.

6.12.4.3 inputPins_

```
const std::vector<std::string>& ModuleRewriter::inputPins_  [private]
```

Definition at line 73 of file verilog_utils.hpp.

6.12.4.4 instance_count_

```
int ModuleRewriter::instance_count_  [private]
```

Definition at line 79 of file verilog_utils.hpp.

6.12.4.5 logger_

```
std::shared_ptr<spdlog::logger> ModuleRewriter::logger_
```

Definition at line 68 of file verilog_utils.hpp.

6.12.4.6 moduleName_

```
const std::string ModuleRewriter::moduleName_  [private]
```

Definition at line 76 of file verilog_utils.hpp.

6.12.4.7 outputPins_

```
const std::vector<std::string>& ModuleRewriter::outputPins_  [private]
```

Definition at line 74 of file verilog_utils.hpp.

6.12.4.8 portInfoMap_

```
std::map<std::string, std::string> ModuleRewriter::portInfoMap_  [private]
```

Definition at line 77 of file verilog_utils.hpp.

The documentation for this class was generated from the following files:

- include/verilog_utils.hpp
- src/verilog_utils.cpp

6.13 PinComparisonResult Struct Reference

```
#include <LogicComparator.hpp>
```

Public Attributes

- std::string pin_name
- std::string ref_expr_raw
- std::string comp_expr_raw
- std::string ref_expr_processed
- std::string comp_expr_processed
- bool comparison_possible = false
- bool are_equivalent = false
- bool ref_compiles = false
- bool comp_compiles = false
- std::optional< Table > ref_truth_table
- std::optional< Table > comp_truth_table
- std::string error_message

6.13.1 Detailed Description

Definition at line 23 of file LogicComparator.hpp.

6.13.2 Member Data Documentation

6.13.2.1 are_equivalent

```
bool PinComparisonResult::are_equivalent = false
```

Definition at line 30 of file LogicComparator.hpp.

6.13.2.2 comp_compiles

```
bool PinComparisonResult::comp_compiles = false
```

Definition at line 32 of file LogicComparator.hpp.

6.13.2.3 comp_expr_processed

```
std::string PinComparisonResult::comp_expr_processed
```

Definition at line 28 of file LogicComparator.hpp.

6.13.2.4 comp_expr_raw

```
std::string PinComparisonResult::comp_expr_raw
```

Definition at line 26 of file LogicComparator.hpp.

6.13.2.5 comp_truth_table

```
std::optional<Table> PinComparisonResult::comp_truth_table
```

Definition at line 34 of file LogicComparator.hpp.

6.13.2.6 comparison_possible

```
bool PinComparisonResult::comparison_possible = false
```

Definition at line 29 of file LogicComparator.hpp.

6.13.2.7 error_message

```
std::string PinComparisonResult::error_message
```

Definition at line 35 of file LogicComparator.hpp.

6.13.2.8 pin_name

```
std::string PinComparisonResult::pin_name
```

Definition at line 24 of file LogicComparator.hpp.

6.13.2.9 ref_compiles

```
bool PinComparisonResult::ref_compiles = false
```

Definition at line 31 of file LogicComparator.hpp.

6.13.2.10 ref_expr_processed

```
std::string PinComparisonResult::ref_expr_processed
```

Definition at line 27 of file LogicComparator.hpp.

6.13.2.11 ref_expr_raw

```
std::string PinComparisonResult::ref_expr_raw
```

Definition at line 25 of file LogicComparator.hpp.

6.13.2.12 ref_truth_table

```
std::optional<Table> PinComparisonResult::ref_truth_table
```

Definition at line 33 of file LogicComparator.hpp.

The documentation for this struct was generated from the following file:

- include/LogicComparator.hpp

6.14 ValuesIterator Class Reference

`#include <Iterators.hpp>`

Public Member Functions

- ValuesIterator (si2drValuesIdT values, si2drErrorT &err)
- ~ValuesIterator ()
- void next ()
- bool end ()

Public Attributes

- si2drValuesIdT values_
- si2drValueTypeT vtype_
- si2drInt32T int_
- si2drFloat64T float_
- si2drStringT str_
- si2drBooleanT bool_
- si2drExprT * exprp_
- si2drErrorT & err_

6.14.1 Detailed Description

Definition at line 37 of file Iterators.hpp.

6.14.2 Constructor & Destructor Documentation

6.14.2.1 ValuesIterator()

```
ValuesIterator::ValuesIterator (
                si2drValuesIdT values,
                si2drErrorT & err )
```

Definition at line 25 of file Iterators.cpp.

6.14.2.2 ~ValuesIterator()

```
ValuesIterator::~ValuesIterator ( )
```

Definition at line 29 of file Iterators.cpp.

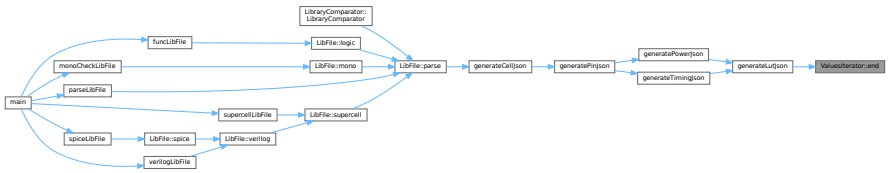
6.14.3 Member Function Documentation

6.14.3.1 end()

```
bool ValuesIterator::end ( )
```

Definition at line 34 of file Iterators.cpp.

Here is the caller graph for this function:

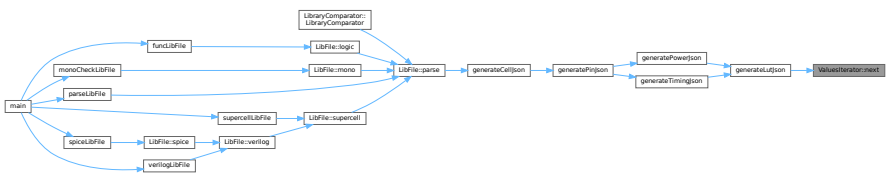


6.14.3.2 next()

```
void ValuesIterator::next ( )
```

Definition at line 31 of file Iterators.cpp.

Here is the caller graph for this function:



6.14.4 Member Data Documentation

6.14.4.1 bool_

```
si2drBooleanT ValuesIterator::bool_
```

Definition at line 49 of file Iterators.hpp.

6.14.4.2 err_

```
si2drErrorT& ValuesIterator::err_
```

Definition at line 51 of file Iterators.hpp.

6.14.4.3 exprp_

```
si2drExprT* ValuesIterator::exprp_
```

Definition at line 50 of file Iterators.hpp.

6.14.4.4 float_

```
si2drFloat64T ValuesIterator::float_
```

Definition at line 47 of file Iterators.hpp.

6.14.4.5 int_

```
si2drInt32T ValuesIterator::int_
```

Definition at line 46 of file Iterators.hpp.

6.14.4.6 str_

```
si2drStringT ValuesIterator::str_
```

Definition at line 48 of file Iterators.hpp.

6.14.4.7 values_

```
si2drValuesIdT ValuesIterator::values_
```

Definition at line 44 of file Iterators.hpp.

6.14.4.8 vtype_

```
si2drValueTypeT ValuesIterator::vtype_
```

Definition at line 45 of file Iterators.hpp.

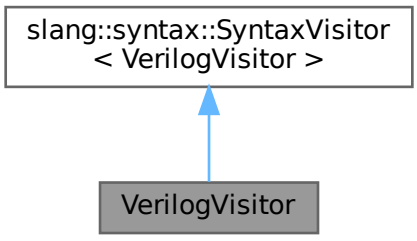
The documentation for this class was generated from the following files:

- include/Iterators.hpp
- src/Iterators.cpp

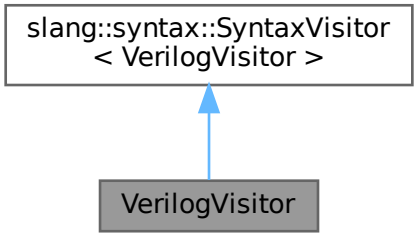
6.15 VerilogVisitor Class Reference

`#include <verilog_utils.hpp>`

Inheritance diagram for VerilogVisitor:



Collaboration diagram for VerilogVisitor:



Public Member Functions

- VerilogVisitor (const std::string &targetCell)
- void handle (const slang::syntax::SyntaxNode &node)
- void handle (const slang::syntax::ModuleDeclarationSyntax &module)
- void handle (const slang::syntax::PortDeclarationSyntax &portDecl)
- void handle (const slang::syntax::HierarchyInstantiationSyntax &hierarchyInst)
- void handle (const slang::syntax::SpecifyBlockSyntax &specifyBlock)

Private Attributes

- const std::string & targetCell_
- int depth_
- bool inTargetModule_

6.15.1 Detailed Description

Definition at line 15 of file verilog_utils.hpp.

6.15.2 Constructor & Destructor Documentation

6.15.2.1 VerilogVisitor()

```
VerilogVisitor::VerilogVisitor (
            const std::string & targetCell )  [inline], [explicit]
```

Definition at line 17 of file verilog_utils.hpp.

6.15.3 Member Function Documentation

6.15.3.1 handle() [1/5]

```
void VerilogVisitor::handle (
            const slang::syntax::HierarchyInstantiationSyntax & hierarchyInst )
```

Definition at line 78 of file verilog_utils.cpp.

6.15.3.2 handle() [2/5]

```
void VerilogVisitor::handle (
            const slang::syntax::ModuleDeclarationSyntax & module )
```

Definition at line 17 of file verilog_utils.cpp.

6.15.3.3 handle() [3/5]

```
void VerilogVisitor::handle (
            const slang::syntax::PortDeclarationSyntax & portDecl )
```

Definition at line 46 of file verilog_utils.cpp.

6.15.3.4 handle() [4/5]

```
void VerilogVisitor::handle (
                const slang::syntax::SpecifyBlockSyntax & specifyBlock )
```

Definition at line 258 of file verilog_utils.cpp.

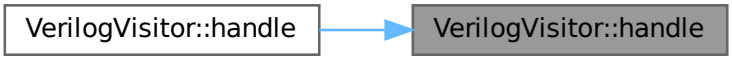
Here is the call graph for this function:

6.15.3.5 handle() [5/5]

```
void VerilogVisitor::handle (
                const slang::syntax::SyntaxNode & node )
```

Definition at line 6 of file verilog_utils.cpp.

Here is the caller graph for this function:



6.15.4 Member Data Documentation

6.15.4.1 depth_

```
int VerilogVisitor::depth_  [private]
```

Definition at line 28 of file verilog_utils.hpp.

6.15.4.2 inTargetModule_

```
bool VerilogVisitor::inTargetModule_  [private]
```

Definition at line 29 of file verilog_utils.hpp.

6.15.4.3 targetCell_

```
const std::string& VerilogVisitor::targetCell_  [private]
```

Definition at line 27 of file verilog_utils.hpp.

The documentation for this class was generated from the following files:

- include/verilog_utils.hpp
- src/verilog_utils.cpp

Chapter 7

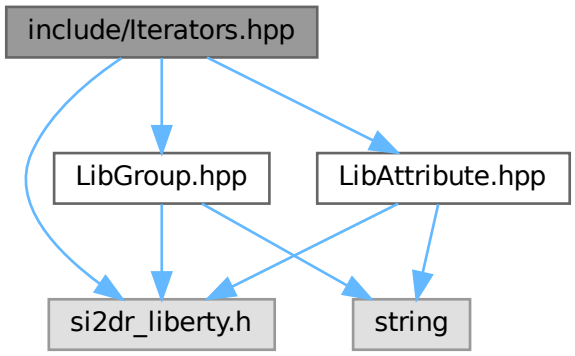
File Documentation

7.1 doc/ChangeLog.md File Reference

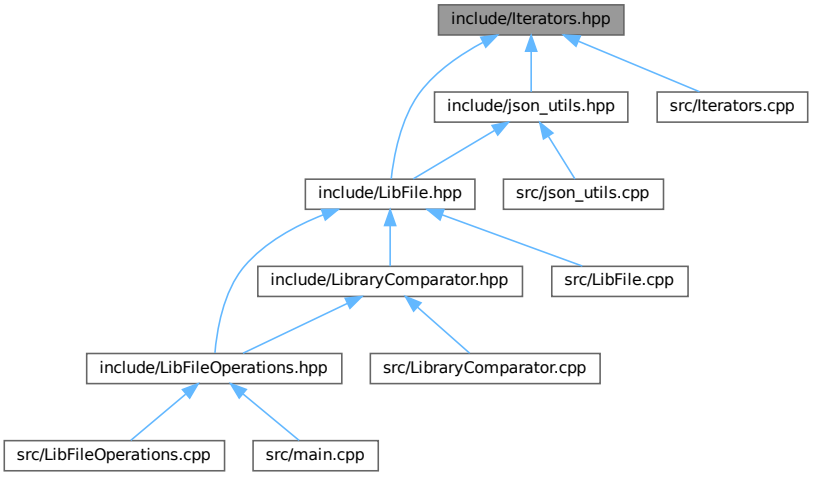
7.2 include/Iterators.hpp File Reference

```
#include "si2dr_liberty.h"
#include "LibAttribute.hpp"
#include "LibGroup.hpp"
```

Include dependency graph for Iterators.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class GroupsIterator
- class AttributesIterator
- class ValuesIterator

7.3 Iterators.hpp

Go to the documentation of this file.

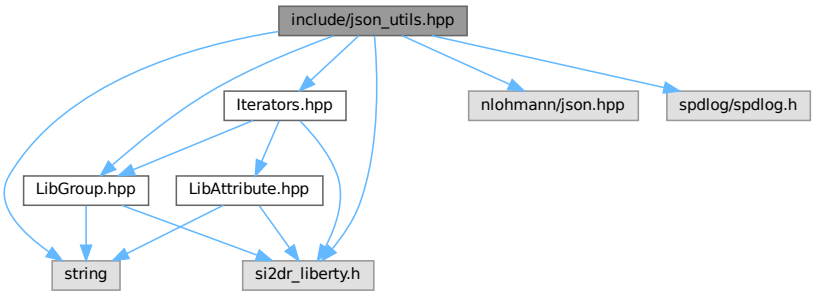
```
00001 #ifndef ITERATORS_H
00002 #define ITERATORS_H
00003
00004 #include "si2dr_liberty.h"
00005
00006 #include "LibAttribute.hpp"
00007 #include "LibGroup.hpp"
00008
00009 class GroupsIterator {
00010 public:
00011   GroupsIterator(si2drGroupsIdT groups, si2drErrorT &err);
00012   ~GroupsIterator();
00013   void next();
00014   bool end();
00015   LibGroup get();
00016
00017 private:
00018   si2drGroupsIdT groups_;
00019   si2drGroupIdT group_;
00020   si2drErrorT &err_;
00021 };
00022
00023 class AttributesIterator {
00024 public:
```

```
00025   AttributesIterator(si2drAttrsIdT attrs, si2drErrorT &err);
00026   ~AttributesIterator();
00027   void next();
00028   bool end();
00029   LibAttribute get();
00030
00031 private:
00032   si2drAttrsIdT attrs_;
00033   si2drAttrIdT attr_;
00034   si2drErrorT &err_;
00035 };
00036
00037 class ValuesIterator {
00038 public:
00039   ValuesIterator(si2drValuesIdT values, si2drErrorT &err);
00040   ~ValuesIterator();
00041   void next();
00042   bool end();
00043
00044   si2drValuesIdT values_;
00045   si2drValueTypeT vtype_;
00046   si2drInt32T int_;
00047   si2drFloat64T float_;
00048   si2drStringT str_;
00049   si2drBooleanT bool_;
00050   si2drExprT *exprp_;
00051   si2drErrorT &err_;
00052 };
00053
00054 #endif // ITERATORS_H
```

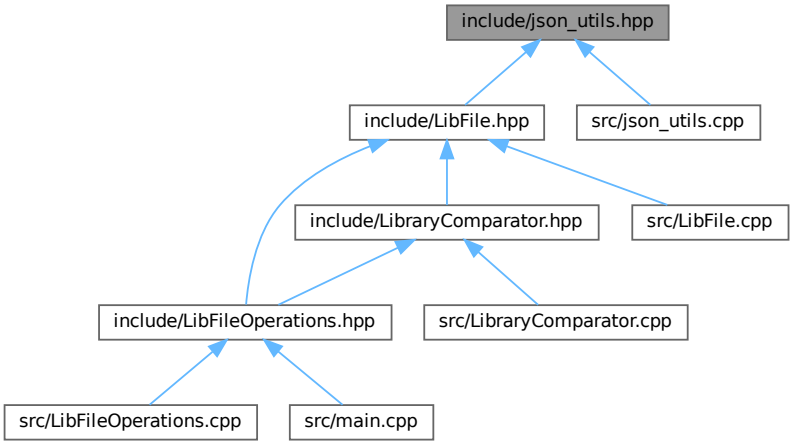
7.4 include/json_utils.hpp File Reference

```
#include <string>
#include "nlohmann/json.hpp"
#include "si2dr_liberty.h"
#include "spdlog/spdlog.h"
#include "Iterators.hpp"
#include "LibGroup.hpp"
```

Include dependency graph for json_utils.hpp:



This graph shows which files directly or indirectly include this file:



Typedefs

- using json = nlohmann::json

Functions

- json generateLutJson (LibGroup &lib_lut_group, si2drErrorT &err)

  Generates a JSON object representation of a look-up table (LUT) from a LibGroup object.
- json generatePowerJson (LibGroup &lib_power_group, si2drErrorT &err)

  Converts a Liberty power group into a JSON representation.
- std::pair< std::string, json > generatePinJson (LibGroup &lib_pin_group, si2drErrorT &err)

  Generates a JSON representation of a Liberty pin group.
- json generateCellJson (LibGroup &lib_cell_group, si2drErrorT &err)

  Generates a JSON representation of a cell from a LibGroup object.

7.4.1 Typedef Documentation

7.4.1.1 json

```
using json = nlohmann::json
```

Definition at line 13 of file json_utils.hpp.

7.4.2 Function Documentation

7.4.2.1 generateCellJson()

```
json generateCellJson (
                LibGroup & lib_cell_group,
                si2drErrorT & err )
```

Generates a JSON representation of a cell from a LibGroup object.

This function processes a LibGroup representing a cell and converts it into a JSON object. It extracts the cell name and processes specific attributes such as area, cell_leakage_power, and cell_footprint. It also processes pins within the cell by categorizing them based on their direction (input, output, internal, inout).

Parameters

lib_cell_group	The LibGroup object representing the cell
err	Reference to a si2drErrorT object for error handling

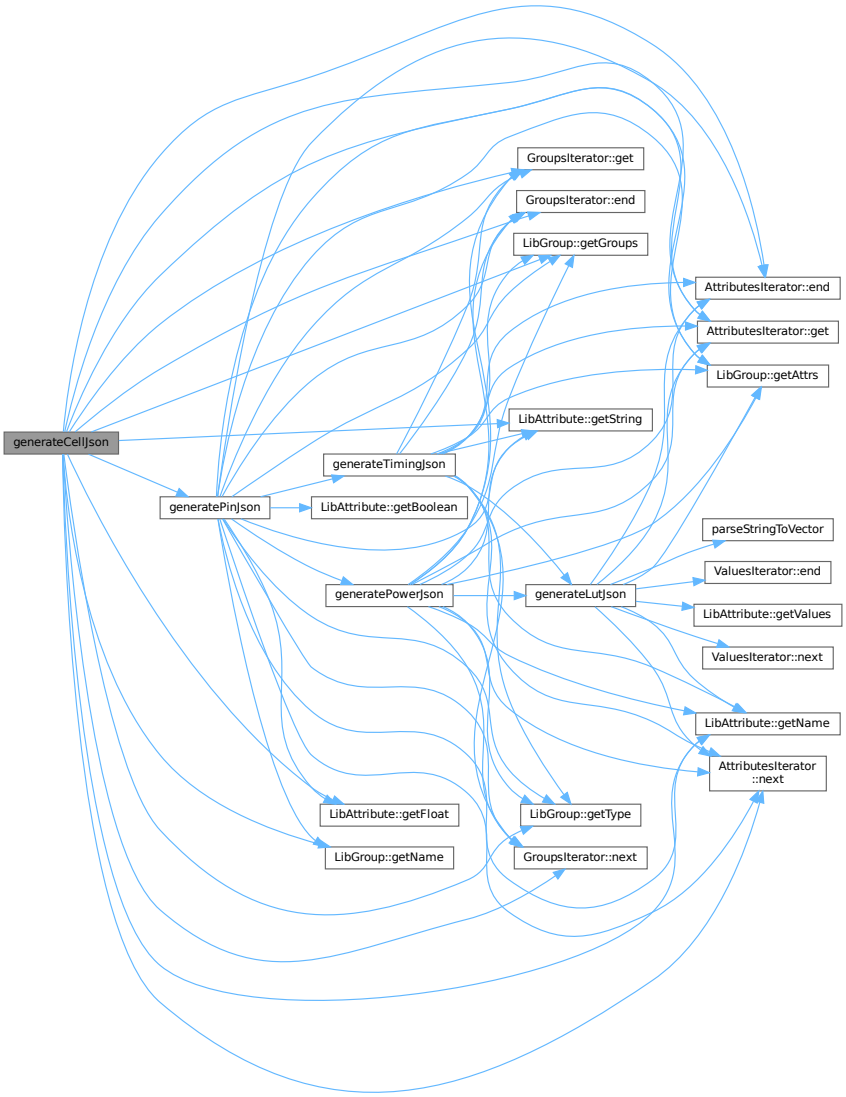
Returns

json A JSON object containing the cell's information with the following structure:

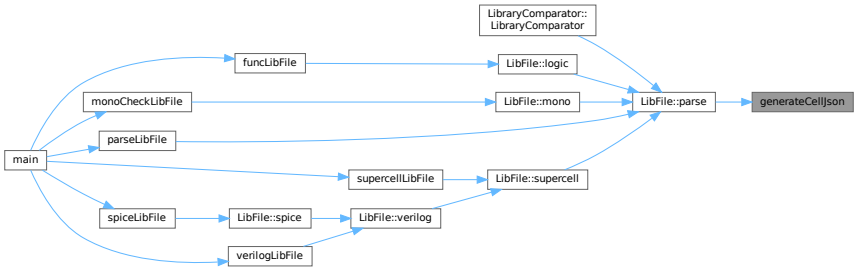
- "cell_name": string - name of the cell
- "area": float (optional) - cell area if present
- "cell_leakage_power": float (optional) - cell leakage power if present
- "cell_footprint": string (optional) - cell footprint if present
- "input_pins": array - list of input pins
- "output_pins": array - list of output pins
- "internal_pins": array - list of internal pins
- "inout_pins": array - list of inout pins

Definition at line 271 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.4.2.2 generateLutJson()

```
json generateLutJson (
                LibGroup & lib_lut_group,
                si2drErrorT & err )
```

Generates a JSON object representation of a look-up table (LUT) from a LibGroup object.

This function iterates through the attributes of the provided LibGroup object representing a LUT and converts them into a JSON structure. It specifically handles the following attributes:

- "index_1": Converted to a vector and stored in the JSON
- "index_2": Converted to a vector and stored in the JSON
- "values": Each value is parsed into a vector and added to an array in the JSON

Any other attributes encountered will trigger a warning message.

Parameters

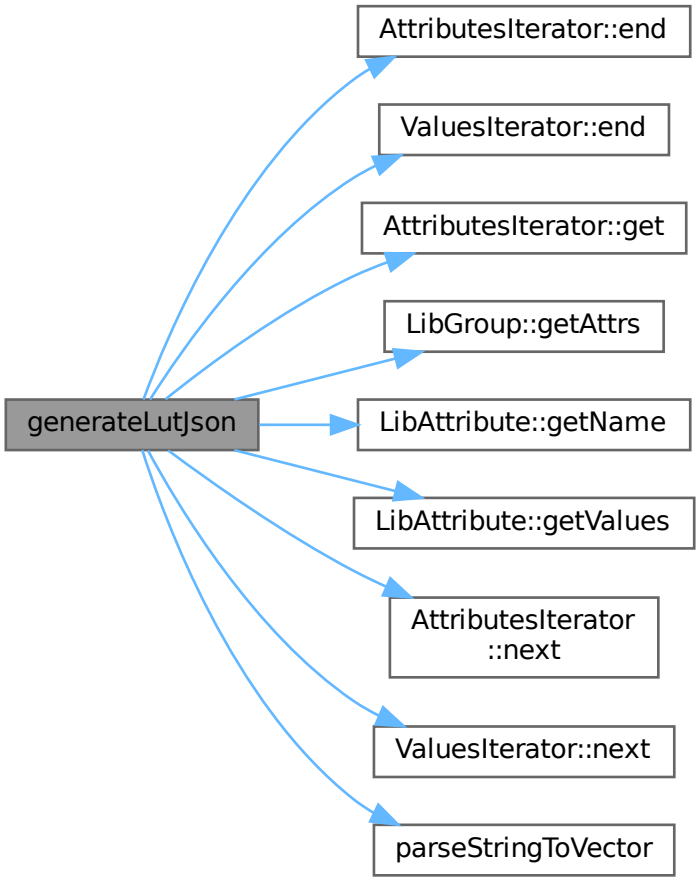
lib_lut_group	The LibGroup object containing the LUT data to be converted
err	Reference to an si2drErrorT object to track any errors during processing

Returns

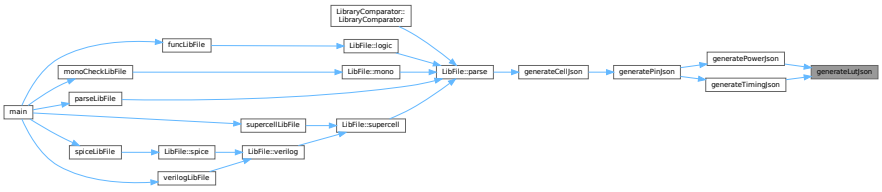
json A JSON object representing the LUT data with keys for "index_1", "index_2", and "values"

Definition at line 60 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.4.2.3 generatePinJson()

```
std::pair< std::string, json > generatePinJson (
```

```
LibGroup & lib_pin_group,
si2drErrorT & err )
```

Generates a JSON representation of a Liberty pin group.

This function traverses a Liberty pin group, extracts relevant attributes and sub-groups, and converts them into a JSON object. It also determines the pin's direction.

Processes the following pin attributes:

- direction
- max_transition, capacitance, rise_capacitance, fall_capacitance, max_capacitance (float types)
- function, power_down_function, related_ground_pin, related_power_pin, three_state (string types)
- clock (boolean type)

Handles the following sub-groups:

- internal_power: Converted using generatePowerJson()
- timing: Converted using generateTimingJson()

Parameters

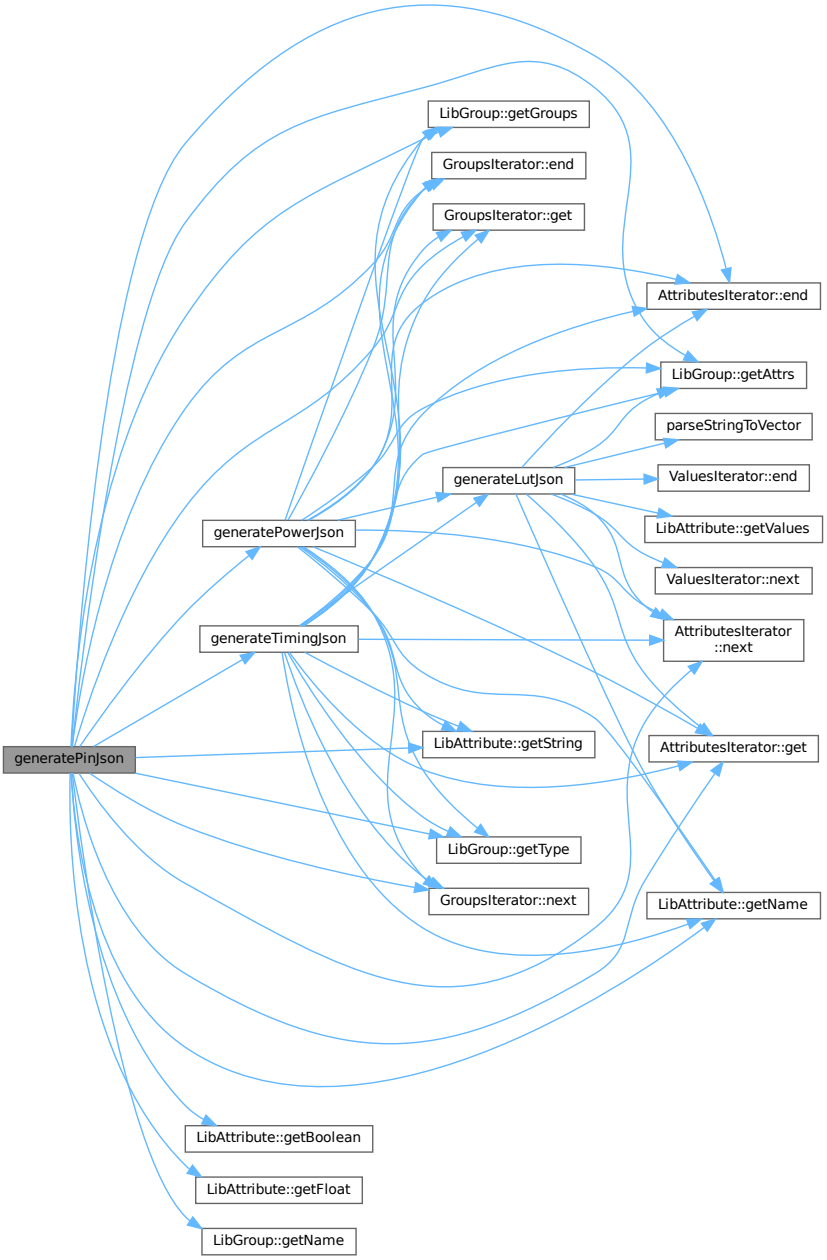
lib_pin_group	The Liberty pin group to process
err	Reference to an error object for tracking Liberty API errors

Returns

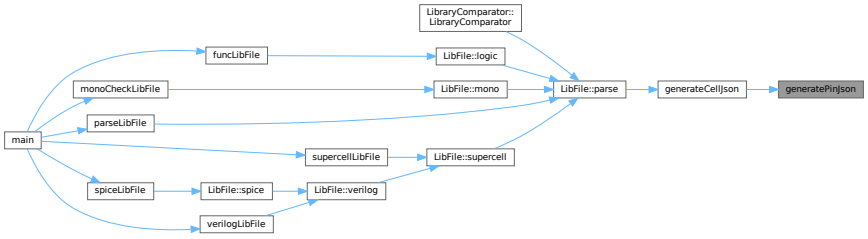
A pair containing the pin direction (string) and the JSON representation of the pin

Definition at line 205 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.4.2.4 generatePowerJson()

```
json generatePowerJson (
                LibGroup & lib_power_group,
                si2drErrorT & err )
```

Converts a Liberty power group into a JSON representation.

This function processes a Liberty power group and converts its attributes and subgroups into a JSON object. It handles attributes like "when", "related_pin", and "related_pg_pin", as well as "rise_power" and "fall_power" subgroups.

Parameters

lib_power_group	The Liberty power group to convert
err	Reference to an si2drErrorT object for error handling

Returns

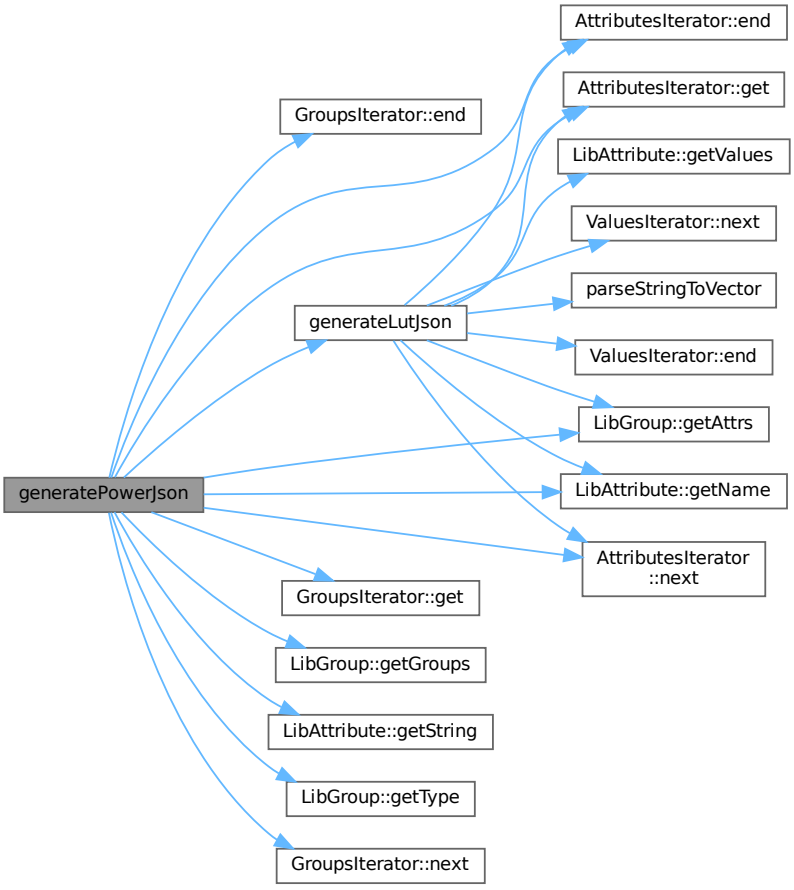
json A JSON object representing the power group data

The function:

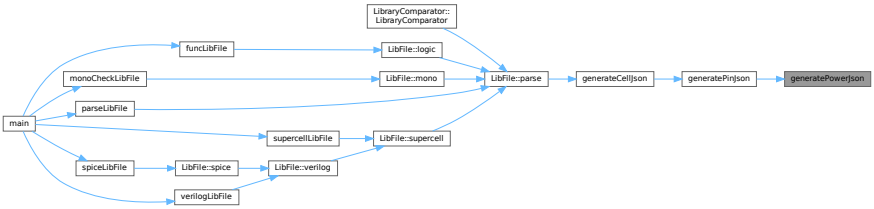
- Extracts string attributes (when, related_pin, related_pg_pin)
- Processes rise_power and fall_power subgroups by converting them to LUT JSON format
- Logs warnings for unknown power subgroup types

Definition at line 152 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.5 json_utils.hpp

Go to the documentation of this file.

```
00001 #ifndef JSON_UTILS_HPP
00002 #define JSON_UTILS_HPP
```

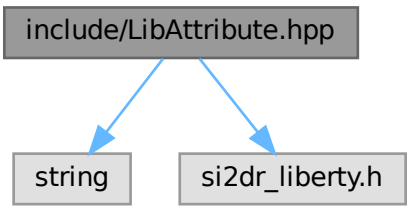
```
00003
00004 #include <string>
00005
00006 #include "nlohmann/json.hpp"
00007 #include "si2dr_liberty.h"
00008 #include "spdlog/spdlog.h"
00009
00010 #include "Iterators.hpp"
00011 #include "LibGroup.hpp"
00012
00013 using json = nlohmann::json;
00014
00015 json generateLutJson(LibGroup &lib_lut_group, si2drErrorT &err);
00016 json generatePowerJson(LibGroup &lib_power_group, si2drErrorT &err);
00017 std::pair<std::string, json> generatePinJson(LibGroup &lib_pin_group, si2drErrorT &err);
00018 json generateCellJson(LibGroup &lib_cell_group, si2drErrorT &err);
00019
00020 #endif // JSON_UTILS_HPP
```

7.6 include/LibAttribute.hpp File Reference

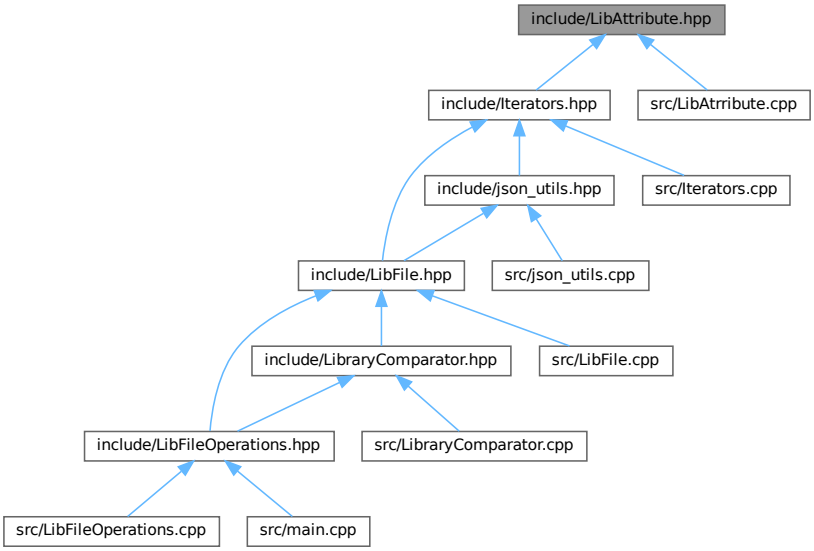
`#include <string>`

`#include "si2dr_liberty.h"`

Include dependency graph for LibAttribute.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class LibAttribute

7.7 LibAttribute.hpp

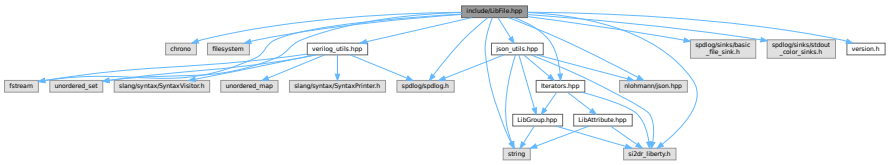
Go to the documentation of this file.

```
00001 #ifndef LIB_ATTRIBUTE_H
00002 #define LIB_ATTRIBUTE_H
00003
00004 #include <string>
00005
00006 #include "si2dr_liberty.h"
00007
00008 class LibAttribute {
00009 public:
00010   LibAttribute(si2drAttrIdT attr, si2drErrorT &err);
00011   ~LibAttribute();
00012   std::string getName();
00013   bool isComplex();
00014   si2drValuesIdT getValues();
00015   long int getInt();
00016   double getFloat();
00017   std::string getString();
00018   bool getBoolean();
00019
00020 private:
00021   si2drAttrIdT attr_;
00022   si2drErrorT &err_;
00023 };
00024
00025 #endif // LIB_ATTRIBUTE_H
```

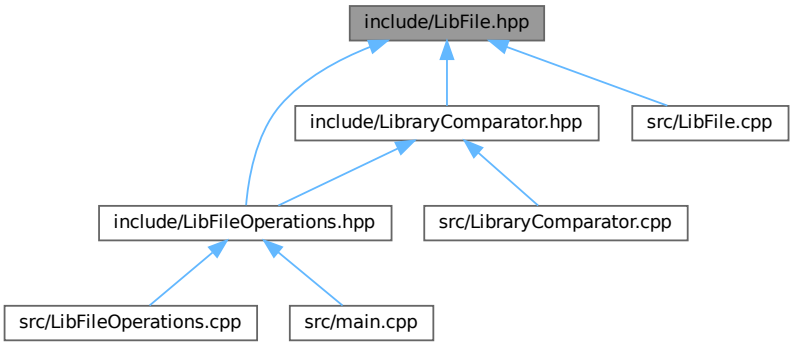
7.8 include/LibFile.hpp File Reference

```
#include <chrono>
#include <filesystem>
#include <fstream>
#include <string>
#include <unordered_set>
#include "nlohmann/json.hpp"
#include "si2dr_liberty.h"
#include "spdlog/sinks/basic_file_sink.h"
#include "spdlog/sinks/stdout_color_sinks.h"
#include "spdlog/spdlog.h"
#include "Iterators.hpp"
#include "json_utils.hpp"
#include "verilog_utils.hpp"
#include "version.h"
```

Include dependency graph for LibFile.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class LibFile

Typedefs

- using json = nlohmann::json

7.8.1 Typedef Documentation

7.8.1.1 json

```
using json = nlohmann::json
```

Definition at line 21 of file LibFile.hpp.

7.9 LibFile.hpp

Go to the documentation of this file.

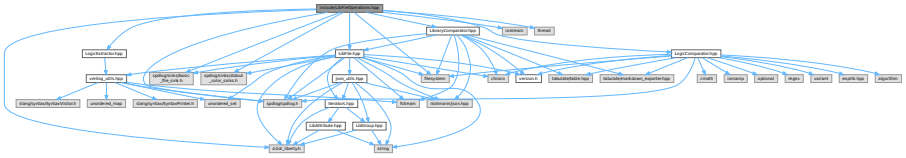
```
00001 #ifndef LIB_FILE_H
00002 #define LIB_FILE_H
00003
00004 #include <chrono>
00005 #include <filesystem>
00006 #include <fstream>
00007 #include <string>
00008 #include <unordered_set>
00009
00010 #include "nlohmann/json.hpp"
00011 #include "si2dr_liberty.h"
00012 #include "spdlog/sinks/basic_file_sink.h"
00013 #include "spdlog/sinks/stdout_color_sinks.h"
00014 #include "spdlog/spdlog.h"
00015
00016 #include "Iterators.hpp"
00017 #include "json_utils.hpp"
00018 #include "verilog_utils.hpp"
00019 #include "version.h"
00020
00021 using json = nlohmann::json;
00022
00023 class LibFile {
00024 public:
00025   LibFile(const std::string &filepath, const std::string &loggername);
00026   ~LibFile();
00027   std::shared_ptr<spdlog::logger> logger_;
00028   std::filesystem::path filepath_; // full path to the file
00029   std::string basename_;           // file name without extension
00030   std::string filename_;           // full file name with extension
00031   std::string libname_ = "";       // library name obtained through parsing
00032   std::string jsonname_ = "";      // json file name to store parsed data
00033   std::string loggername_ = "";    // log file name
00034   json lib_json_ = json::object();
00035   void writeJsonToFile();
00036   void parse();
00037   void modify();
00038   void mono(const bool is_slew);
```

```
00039   void supercell(const int chain_length, const std::vector<std::string> &cell_names);
00040   void verilog(const int chain_length, const std::vector<std::string> &cell_names);
00041   void spice(const int chain_length, const std::vector<std::string> &cell_names,
00042              const std::string &verilog_lib_file, const std::string &spice_lib_file);
00043   std::map<std::string, std::string> logic(const std::string &cell_name);
00044
00045 private:
00046   si2drErrorT err_;
00047   int process_ = 0;
00048   float voltage_ = 0.0;
00049   int temperature_ = 0;
00050   void read();
00051   bool checkTimingArcMonotonicity(const json &cell, const json &pin, const json &arc,
00052                                   const std::string &type, bool is_slew);
00053   // --- Private Helper Methods ---
00054
00055   // Helper function to split a string by whitespace
00056   std::vector<std::string> splitString(const std::string &s);
00057
00058   // Function to generate RC lines based on instance index and stage
00059   void generateRCLines(std::ofstream &outFile, const std::string &netName, int instanceIndex,
00060                        bool isFinalStage);
00061
00062   // Function to modify the SPICE netlist generated by v2lvs
00063   bool modifySpiceNetlist(const std::string &v2lvsSpiceFile, // Input is the v2lvs output
00064                           const std::string &finalSpiceFile, // Output is the final file
00065                           const std::string &targetGlobalLine);
00066 };
00067
00068 #endif // LIB_FILE_H
```

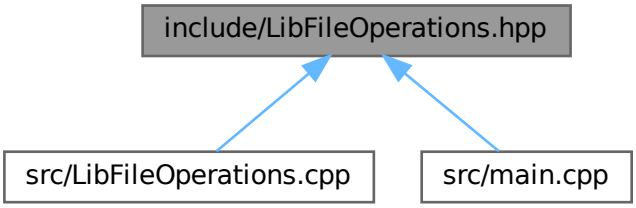
7.10 include/LibFileOperations.hpp File Reference

```
#include <filesystem>
#include <iostream>
#include <thread>
#include "si2dr_liberty.h"
#include "spdlog/sinks/basic_file_sink.h"
#include "spdlog/sinks/stdout_color_sinks.h"
#include "spdlog/spdlog.h"
#include "LibFile.hpp"
#include "LibraryComparator.hpp"
#include "LogicComparator.hpp"
#include "LogicExtractor.hpp"
```

Include dependency graph for LibFileOperations.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- void printInfo ()

  Sets up and configures the global logger and prints application information.
- void parseLibFile (const std::string &library_path, const std::string log_file_name)

  Parses a library file and generates corresponding output.
- void monoCheckLibFile (const std::string &library_path, const std::string log_file_name, bool is↩ _slew)

  Performs monotonicity check on a library file.
- void supercellLibFile (const std::string &library_path, const std::string &log_file_name, int chain↩ _length, const std::vector< std::string > &cell_names)

  Creates supercell map structures from a Liberty library file.
- void verilogLibFile (const std::string &library_path, const std::string &log_file_name, int chain↩ _length, const std::vector< std::string > &cell_names)

  Generates Verilog files from a library file for specified cells.
- void spiceLibFile (const std::string &library_path, const std::string &log_file_name, int chain↩ _length, const std::vector< std::string > &cell_names, const std::string &verilog_lib_file, const std::string &spice_lib_file)

  Generates a SPICE library file from a given library file, applying a specified chain length and cell names.
- void compareLibFiles (const std::string &ref_lib, const std::string &comp_lib, const double reltol, const double abstol, std::string &report_file_name)

  Compares two library files and generates a detailed comparison report.
- void funcLibFile (const std::string &ref_file, const std::string &comp_file, const std::vector< std↩ ::string > &cell_names, std::string &report_file_name)

  Performs a functional equivalence check between two files (Liberty or Verilog) for a given set of cells.

7.10.1 Function Documentation

7.10.1.1 compareLibFiles()

```
void compareLibFiles (
                const std::string & ref_lib,
                const std::string & comp_lib,
                const double reltol,
                const double abstol,
                std::string & report_file_name )
```

Compares two library files and generates a detailed comparison report.

This function compares a reference library file with another library file, performing validation checks based on specified tolerance parameters. The comparison results are written to a report file in markdown or text format.

Parameters

ref_lib	Path to the reference library file to use as the baseline
comp_lib	Path to the library file to compare against the reference
reltol	Relative tolerance for numerical comparisons (must be >= 0.0)
abstol	Absolute tolerance for numerical comparisons
report_file_name	[in,out] Name of the file to write the comparison report to. If empty, defaults to [comp_lib_basename].cmp.md. If provided but doesn't end with .txt or .md, .md will be appended.

Note

The function will log an error and return without comparing if reltol is invalid.

Log files will be created with the naming pattern [library_basename].cmp.log

The function uses the LibraryComparator class to perform the actual comparison.

Definition at line 249 of file LibFileOperations.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

7.10.1.2 funcLibFile()

```
void funcLibFile (
                const std::string & ref_file,
                const std::string & comp_file,
                const std::vector< std::string > & cell_names,
                std::string & report_file_name )
```

Performs a functional equivalence check between two files (Liberty or Verilog) for a given set of cells.

This function compares the logic functions of specified cells in two files, which can be either in Liberty (.lib) or Verilog (.v) format. It extracts the logic functions for each cell from both files, compares them, and generates a report summarizing the comparison results.

Parameters

ref_file	The path to the reference file (Liberty or Verilog).
comp_file	The path to the comparison file (Liberty or Verilog).
cell_names	A vector of cell names to be checked for functional equivalence.
report_file_name	A string to store the name of the report file. If empty, a default name is generated. If the provided name does not end with ".txt" or ".md", ".md" is appended.

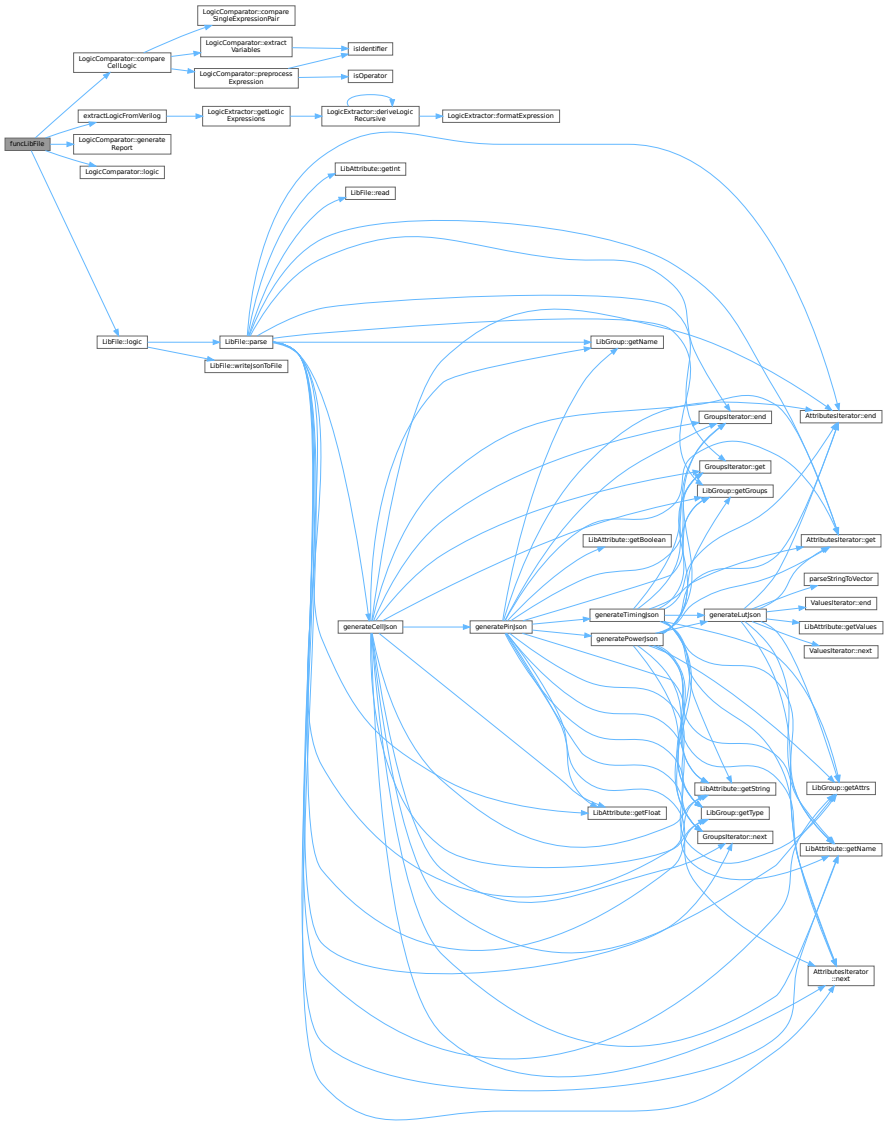
The function first checks the file extensions to determine the file format. It then extracts the logic functions for each specified cell from both files. The logic functions are then compared, and a report is generated, which includes the comparison results for each cell. The report is written to the specified report file.

Note

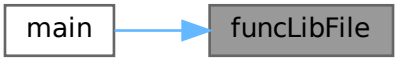
- If no cell names are provided, the function logs an error and returns.
- If the reference or comparison file format is not supported (i.e., not .lib or .v), the function logs an error and returns.
- The report file is cleared before writing the comparison results.
- The function uses spdlog for logging information, warnings, and errors.
- The function utilizes the LogicComparator class to perform the logic comparison and generate the report.
- Memory allocated for LibFile objects is managed using raw pointers and must be manually deallocated to prevent memory leaks.

Definition at line 308 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.1.3 monoCheckLibFile()

```
void monoCheckLibFile (
                const std::string & library_path,
                const std::string log_file_name,
                bool is_slew )
```

Performs monotonicity check on a library file.

This function validates the monotonicity of timing data in a library file. It creates a log file to record the results of the check and handles any exceptions that occur during the process.

Parameters

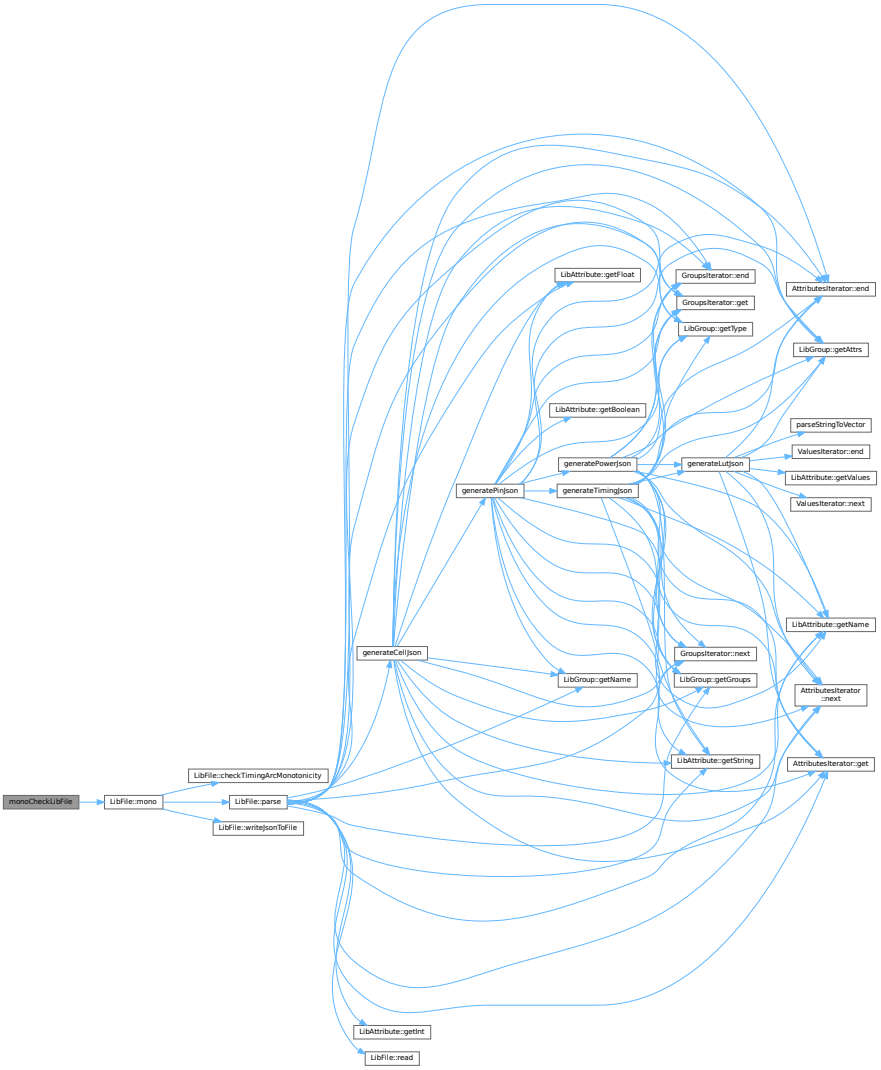
library_path	Path to the library file to check
log_file_name	Name of the log file to create (optional). If empty, a default name will be generated from the library file name
is_slew	Flag indicating whether to check input slew monotonicity (true) or output load monotonicity (false)

Exceptions

The	function catches and logs any exceptions but does not rethrow them

Definition at line 81 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

7.10.1.4 parseLibFile()

```
void parseLibFile (
                const std::string & library_path,
                const std::string log_file_name )
```

Parses a library file and generates corresponding output.

This function processes the given library file, parsing its contents and generating a JSON output. It also logs the parsing process to a specified log file or creates a default log file if none is provided.

Parameters

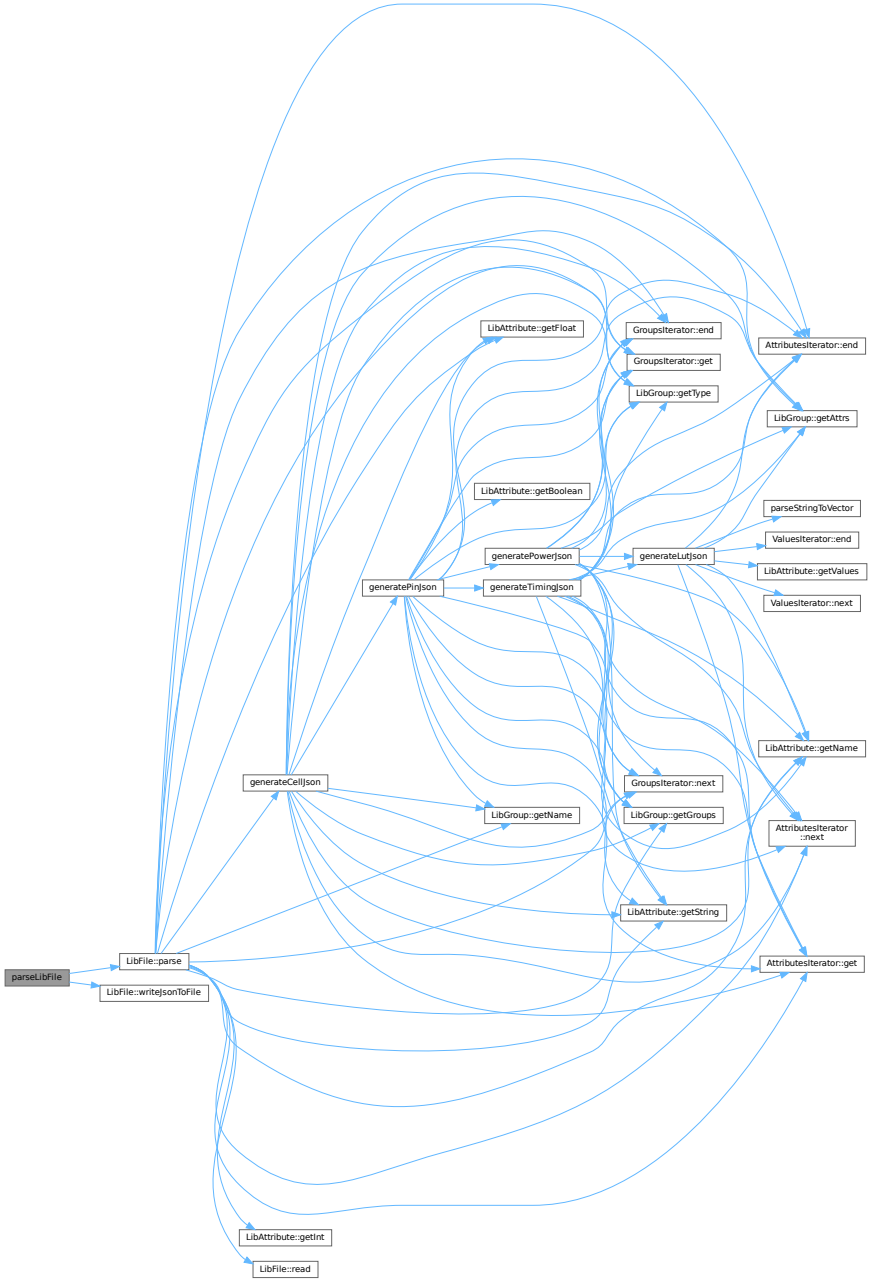
library_path	Path to the library file that needs to be parsed
log_file_name	Optional name for the log file. If empty, a default name is generated based on the library filename with ".parse.log" extension

Exceptions

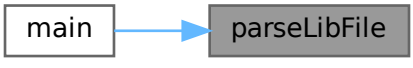
The	function catches and logs any exceptions but doesn't propagate them

Definition at line 49 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.1.5 printInfo()

```
void printInfo ( )
```

Sets up and configures the global logger and prints application information.

This function performs the following operations:

1. Creates a console sink for logging with level set to INFO
2. Creates a file sink for logging with level set to TRACE, saving to [APP_NAME].log
3. Configures a logger with both sinks and sets it as the default logger
4. Outputs basic application information:
    - Version and build timestamp
    - Author information
    - Log file location

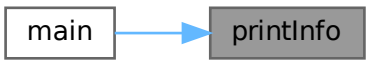
Note

Uses spdlog library for logging functionality

Depends on APP_NAME, APP_VERSION, BUILD_TIMESTAMP, APP_AUTHOR, and APP↩ _CONTACT macros

Definition at line 18 of file LibFileOperations.cpp.

Here is the caller graph for this function:



7.10.1.6 spiceLibFile()

```
void spiceLibFile (
                const std::string & library_path,
                const std::string & log_file_name,
                int chain_length,
                const std::vector< std::string > & cell_names,
                const std::string & verilog_lib_file,
                const std::string & spice_lib_file )
```

Generates a SPICE library file from a given library file, applying a specified chain length and cell names.

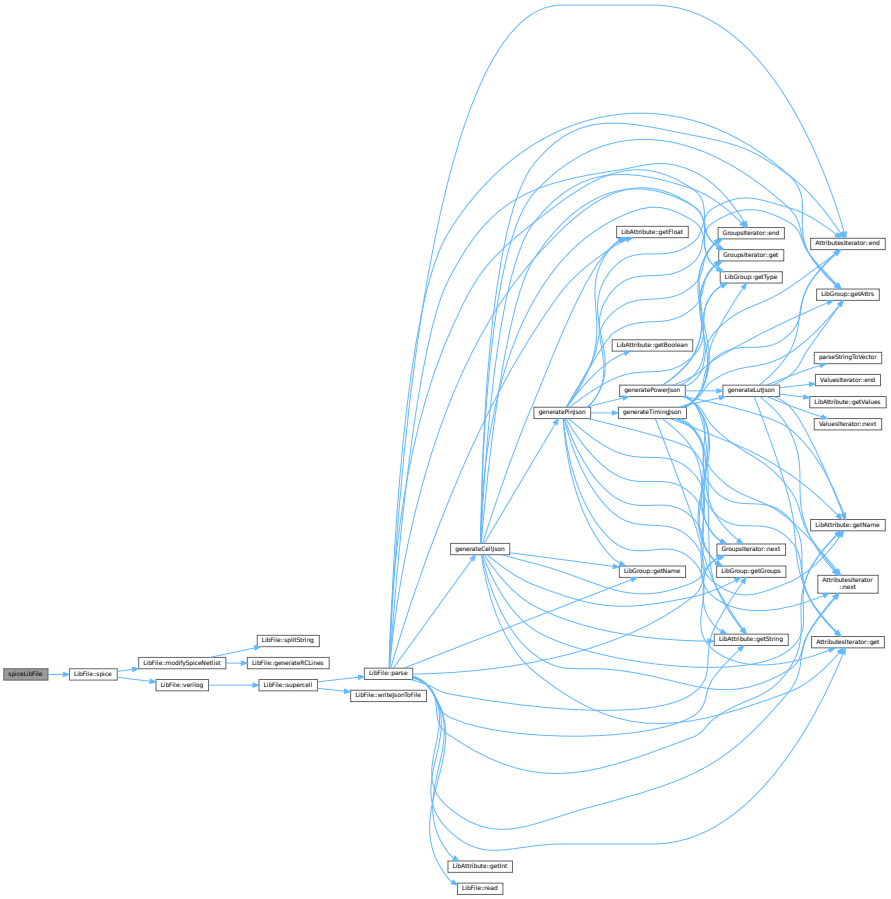
This function takes a library file path, a log file name, a chain length, a vector of cell names, a Verilog library file path, and a SPICE library file path as input. It initializes a LibFile object, validates the chain length, and then calls the spice method of the LibFile object to generate the SPICE library file. It logs the start and end of the SPICE generation process, as well as any errors that occur.

Parameters

library_path	The path to the input library file.
log_file_name	The name of the log file. If empty, a default log file name is generated based on the library file name.
chain_length	The chain length to use during SPICE generation. Must be $>= 1$.
cell_names	A vector of cell names to include in the SPICE generation.
verilog_lib_file	The path to the Verilog library file.
spice_lib_file	The path to the output SPICE library file.

Definition at line 202 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

7.10.1.7 supercellLibFile()

```
void supercellLibFile (
                const std::string & library_path,
                const std::string & log_file_name,
```

```
int chain_length,
const std::vector< std::string > & cell_names )
```

Creates supercell map structures from a Liberty library file.

This function reads a Liberty file, creates supercell structures based on the specified chain length and cell names, and logs the process to a file.

Parameters

library_path	Path to the Liberty file to process
log_file_name	Name of the log file (if empty, defaults to "[library_name].supercell.log")
chain_length	The length of chains to create (must be >= 1)
cell_names	Vector of cell names to process for supercell generation

Exceptions

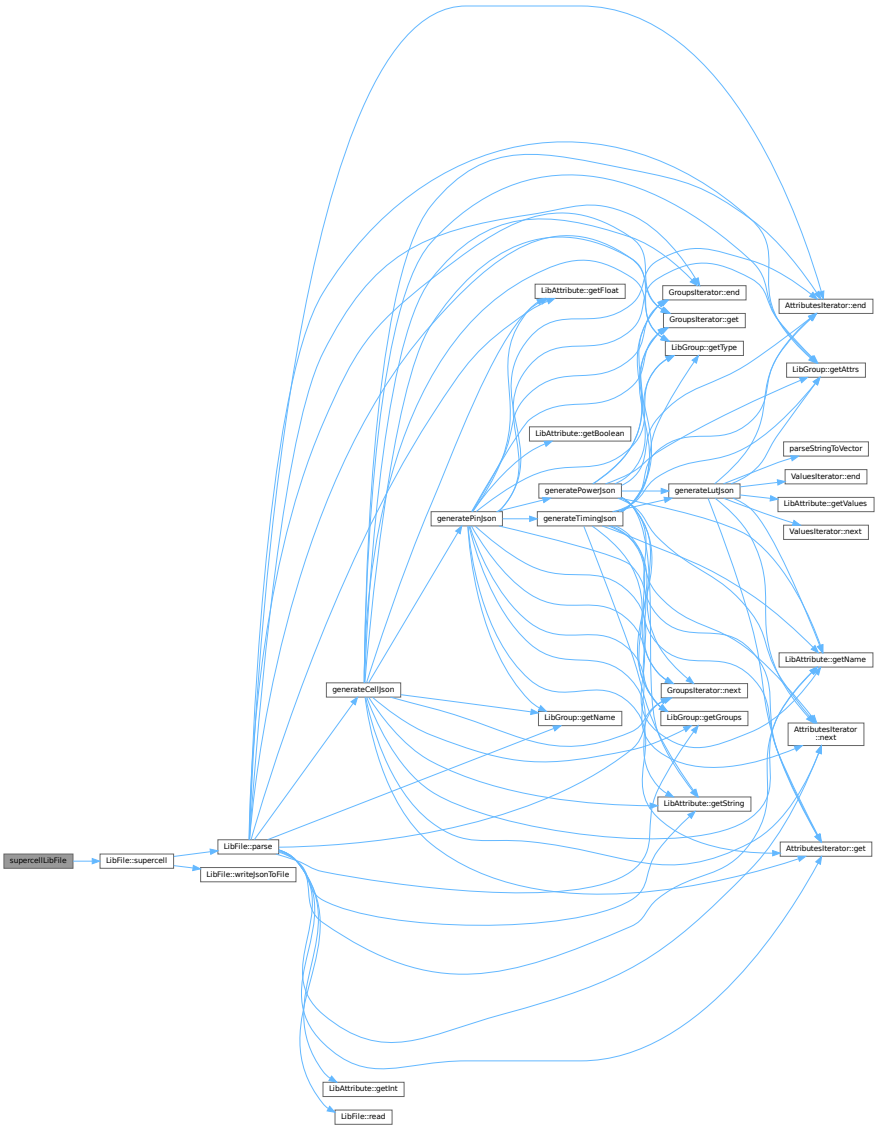
May	pass through exceptions from the LibFile::supercell method

Note

The function validates the chain length and logs all activities including errors that might occur during processing

Definition at line 116 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

7.10.1.8 verilogLibFile()

```
void verilogLibFile (
                const std::string & library_path,
                const std::string & log_file_name,
                int chain_length,
                const std::vector< std::string > & cell_names )
```

Generates Verilog files from a library file for specified cells.

This function processes a library file and generates Verilog representation for the specified cell names with a given chain length. The operation results are logged to a specified or default log file.

Parameters

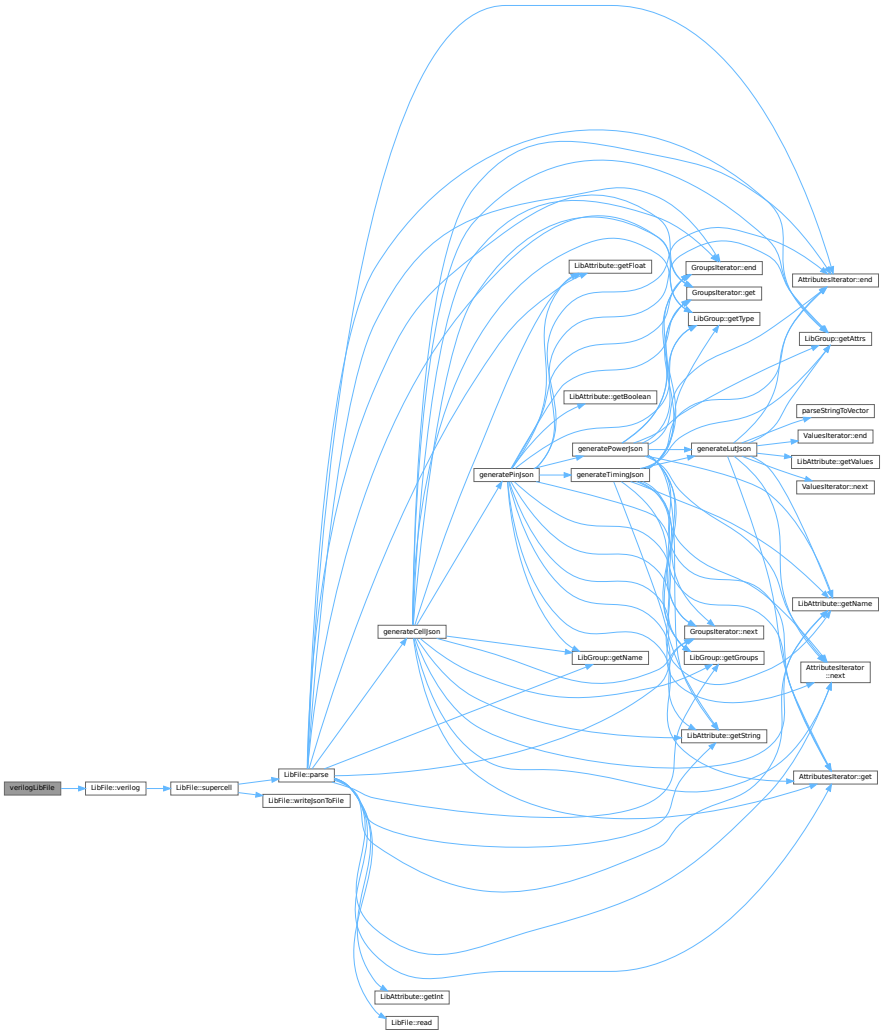
library_path	Path to the library file to process
log_file_name	Name for the log file (if empty, a default name will be generated)
chain_length	Number of cells to chain together, must be >= 1
cell_names	Vector of cell names to generate Verilog for

Exceptions

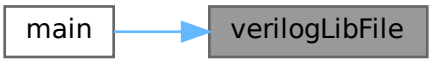
Catches	any exceptions from the verilog generation process and logs them

Definition at line 157 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.11 LibFileOperations.hpp

Go to the documentation of this file.

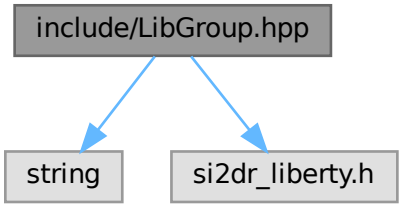
```
00001 #ifndef LIBFILEOPERATIONS_H
00002 #define LIBFILEOPERATIONS_H
00003
00004 #include <filesystem>
00005 #include <iostream>
00006 #include <thread>
00007
00008 #include "si2dr_liberty.h"
00009 #include "spdlog/sinks/basic_file_sink.h"
00010 #include "spdlog/sinks/stdout_color_sinks.h"
00011 #include "spdlog/spdlog.h"
00012
00013 #include "LibFile.hpp"
00014 #include "LibraryComparator.hpp"
00015 #include "LogicComparator.hpp"
00016 #include "LogicExtractor.hpp"
00017
00018 void printInfo();
00019 void parseLibFile(const std::string &library_path, const std::string log_file_name);
00020 void monoCheckLibFile(const std::string &library_path, const std::string log_file_name,
00021                       bool is_slew);
00022 void supercellLibFile(const std::string &library_path, const std::string &log_file_name,
00023                       int chain_length, const std::vector<std::string> &cell_names);
00024 void verilogLibFile(const std::string &library_path, const std::string &log_file_name,
00025                     int chain_length, const std::vector<std::string> &cell_names);
00026 void spiceLibFile(const std::string &library_path, const std::string &log_file_name,
00027                   int chain_length, const std::vector<std::string> &cell_names,
00028                   const std::string &verilog_lib_file, const std::string &spice_lib_file);
00029 void compareLibFiles(const std::string &ref_lib, const std::string &comp_lib, const double reltol,
00030                      const double abstol, std::string &report_file_name);
00031 void funcLibFile(const std::string &ref_file, const std::string &comp_file,
00032                  const std::vector<std::string> &cell_names, std::string &report_file_name);
00033
00034 #endif // LIBFILEOPERATIONS_H
```

7.12 include/LibGroup.hpp File Reference

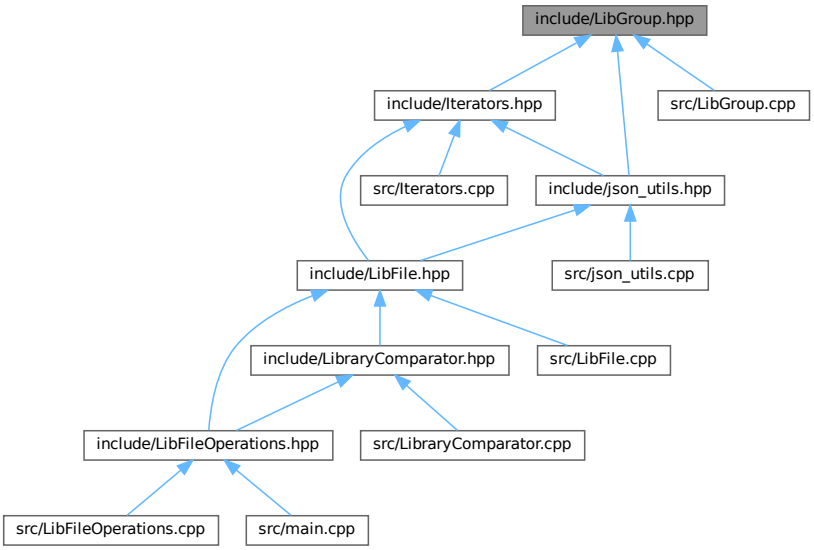
`#include <string>`

`#include "si2dr_liberty.h"`

Include dependency graph for LibGroup.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class LibGroup

7.13 LibGroup.hpp

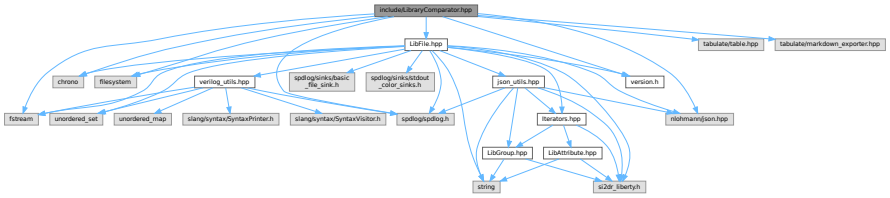
Go to the documentation of this file.

```
00001 #ifndef LIB_GROUP_H
00002 #define LIB_GROUP_H
00003
00004 #include <string>
00005
00006 #include "si2dr_liberty.h"
00007
00008 class LibGroup {
00009 public:
00010   LibGroup(si2drGroupIdT group, si2drErrorT &err);
00011   ~LibGroup();
00012   std::string getName();
00013   std::string getType();
00014   si2drAttrsIdT getAttrs();
00015   si2drGroupsIdT getGroups();
00016
00017 private:
00018   si2drGroupIdT group_;
00019   si2drErrorT &err_;
00020 };
00021
00022 #endif // LIB_GROUP_H
```

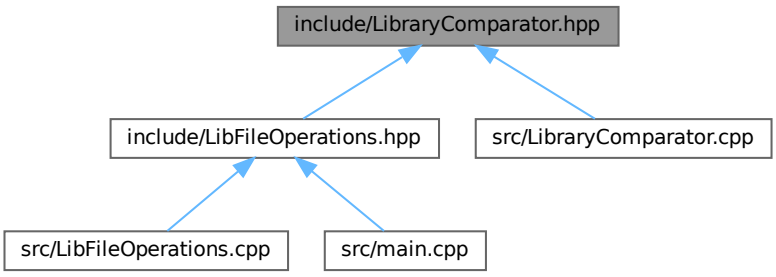
7.14 include/LibraryComparator.hpp File Reference

```
#include <chrono>
#include <filesystem>
#include <fstream>
#include "nlohmann/json.hpp"
#include "spdlog/spdlog.h"
#include "tabulate/table.hpp"
#include <tabulate/markdown_exporter.hpp>
#include "LibFile.hpp"
#include "version.h"
```

Include dependency graph for LibraryComparator.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class LibraryComparator

Typedefs

- using json = nlohmann::json

7.14.1 Typedef Documentation

7.14.1.1 json

```
using json = nlohmann::json
```

Definition at line 16 of file LibraryComparator.hpp.

7.15 LibraryComparator.hpp

Go to the documentation of this file.

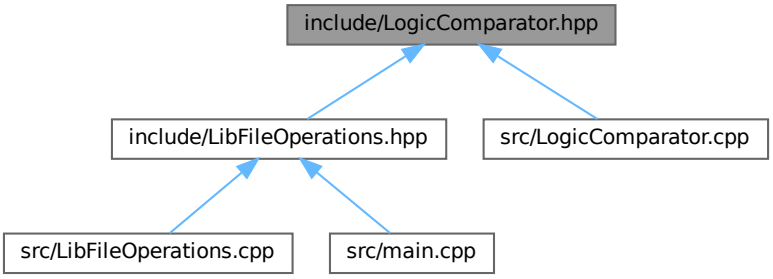
```
00001 #ifndef COMPARE_HPP
00002 #define COMPARE_HPP
00003
00004 #include <chrono>
00005 #include <filesystem>
00006 #include <fstream>
00007
00008 #include "nlohmann/json.hpp"
00009 #include "spdlog/spdlog.h"
00010 #include "tabulate/table.hpp"
00011 #include <tabulate/markdown_exporter.hpp>
00012
00013 #include "LibFile.hpp"
00014 #include "version.h"
00015
00016 using json = nlohmann::json;
00017 using namespace tabulate;
00018
00019 class LibraryComparator {
00020 public:
00021   LibraryComparator(LibFile &ref_libfile, LibFile &comp_libfile, double reltol, double abstol);
00022   std::filesystem::path ref_lib_path_;
00023   std::filesystem::path comp_lib_path_;
00024   void generateReport(const std::string &output_file);
00025
00026 private:
00027   json ref_json_;
00028   json comp_json_;
00029   double reltol_;
00030   double abstol_;
00031
00032   void compareCell(const std::string &cell_name, const json &ref_cell, const json &comp_cell,
00033                    Table &table);
00034   void comparePin(const std::string &cell_name, const std::string &pin_name, const json &ref_pin,
00035                   const json &comp_pin, Table &table);
00036   void compareTimingArc(const std::string &cell_name, const std::string &pin_name,
00037                         const std::string &timing_type, const json &ref_timing_arc,
00038                         const json &comp_timing_arc, Table &table);
00039   void compareLut(const std::string &cell_name, const std::string &pin_name,
00040                   const std::string &timing_type, const std::string &related_pin,
00041                   const std::string &arc_name, const json &ref_lut, const json &comp_lut,
00042                   Table &table);
00043   // void configureTableFormat(Table &table);
00044 };
00045
00046 #endif // COMPARE_HPP
```

7.16 include/LogicComparator.hpp File Reference

```
#include <algorithm>
#include <chrono>
#include <cmath>
#include <filesystem>
#include <iomanip>
#include <optional>
#include <regex>
#include <variant>
#include "exprtk.hpp"
#include "tabulate/markdown_exporter.hpp"
#include "tabulate/table.hpp"
#include <spdlog/spdlog.h>
#include "version.h"
```

Include dependency graph for LogicComparator.hpp:

This graph shows which files directly or indirectly include this file:



Classes

- struct PinComparisonResult
- class LogicComparator

7.17 LogicComparator.hpp

Go to the documentation of this file.

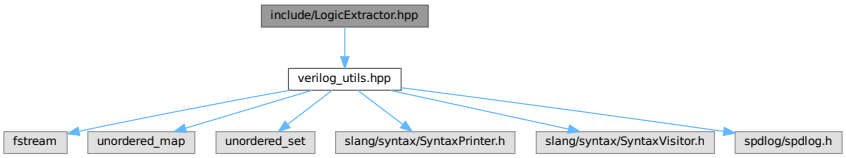
```
00001 #ifndef LOGIC_COMPARATOR_HPP
00002 #define LOGIC_COMPARATOR_HPP
00003
00004 #include <algorithm>
00005 #include <chrono>
00006 #include <cmath>
00007 #include <filesystem>
00008 #include <iomanip>
00009 #include <optional> // To store tables optionally
00010 #include <regex>    // For regular expressions
00011 #include <variant>
00012
00013 #include "exprtk.hpp"
00014 #include "tabulate/markdown_exporter.hpp"
00015 #include "tabulate/table.hpp"
00016 #include <spdlog/spdlog.h>
00017
00018 #include "version.h"
00019
00020 using namespace tabulate;
00021
00022 // Structure to hold results for a single pin comparison
00023 struct PinComparisonResult {
00024   std::string pin_name;
00025   std::string ref_expr_raw;
00026   std::string comp_expr_raw;
00027   std::string ref_expr_processed;
00028   std::string comp_expr_processed;
00029   bool comparison_possible = false; // Was comparison attempted?
00030   bool are_equivalent = false;
00031   bool ref_compiles = false;
00032   bool comp_compiles = false;
00033   std::optional<Table> ref_truth_table; // Store tables only if needed/successful
00034   std::optional<Table> comp_truth_table;
00035   std::string error_message; // Store any error during comparison
00036 };
00037
00038 class LogicComparator {
00039 public:
00040   LogicComparator(const std::map<std::string, std::string> &ref_outpin_map,
00041                   const std::map<std::string, std::string> &comp_outpin_map,
00042                   const std::string &cell_name);
00043
00044   // Example code from exprtk documentation
00045   void logic();
00046
00047   // Preprocessing function
00048   std::string preprocessExpression(const std::string &input_expr);
00049
00050   // Helper to extract unique sorted variables from TWO expressions
00051   bool extractVariables(const std::string &expr1, const std::string &expr2,
00052                         std::vector<std::string> &sorted_vars);
00053
00064   void compareSingleExpressionPair(const std::string &ref_expression_processed,
00065                                    const std::string &comp_expression_processed,
00066                                    const std::vector<std::string> &sorted_vars,
00067                                    PinComparisonResult &result); // Pass result struct
00068
00073   void compareCellLogic();
00074
00080   void generateReport(const std::string &output_file);
00081
00082 private:
00083   std::map<std::string, std::string> ref_outpin_map_;
```

```
00084   std::map<std::string, std::string> comp_outpin_map_;
00085   std::string cell_name_;
00086   std::map<std::string, PinComparisonResult> all_pin_results_;
00087 };
00088
00089 #endif // LOGIC_COMPARATOR_HPP
```

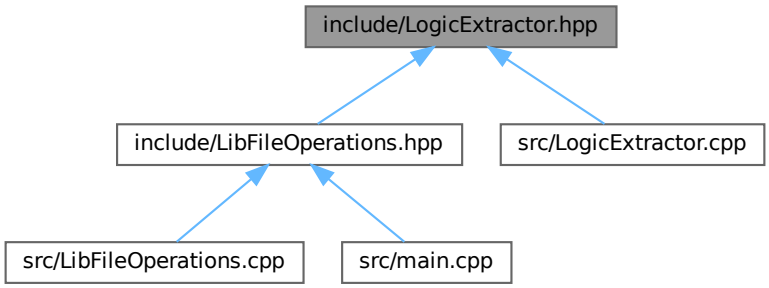
7.18 include/LogicExtractor.hpp File Reference

`#include "verilog_utils.hpp"`

Include dependency graph for LogicExtractor.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct GateInfo
- class LogicExtractor

Functions

- void extractAndPrintNetlistInfo (const std::string &verilog_file, const std::string &cell)

  Extracts and prints netlist information from a Verilog file for a specified cell.

- std::map< std::string, std::string > extractLogicFromVerilog (const std::string &verilog_file, const std::string &cell)

  Extracts logic expressions from a Verilog file for a specified cell.

7.18.1 Function Documentation

7.18.1.1 extractAndPrintNetlistInfo()

```
void extractAndPrintNetlistInfo (
                const std::string & verilog_file,
                const std::string & cell )
```

Extracts and prints netlist information from a Verilog file for a specified cell.

This function parses a Verilog file using the slang library, extracts information about the primary inputs, primary outputs, internal wires, and gate drivers within a specified cell (module). It then prints a summary of the extracted information to the console using spdlog.

Parameters

verilog_file	The path to the Verilog file to be parsed.
cell	The name of the cell (module) for which to extract netlist information.

Exceptions

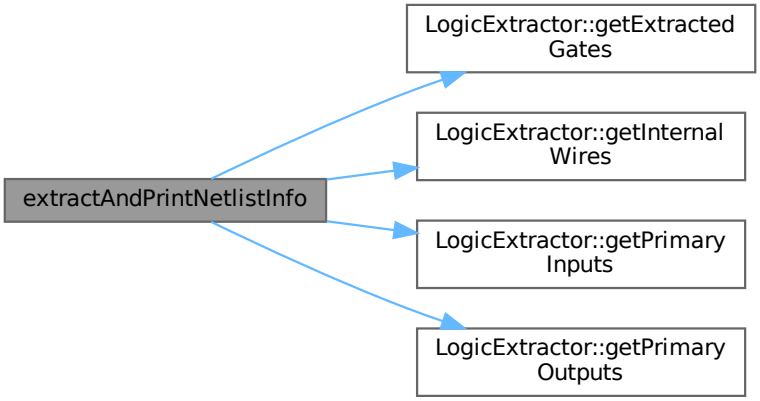
std::exception	If any error occurs during Verilog parsing or info extraction.

Note

The function uses the slang library for Verilog parsing and a custom LogicExtractor class to extract the desired information. Error messages are logged using spdlog.

Definition at line 676 of file LogicExtractor.cpp.

Here is the call graph for this function:



7.18.1.2 extractLogicFromVerilog()

```
std::map< std::string, std::string > extractLogicFromVerilog (
                const std::string & verilog_file,
                const std::string & cell )
```

Extracts logic expressions from a Verilog file for a specified cell.

This function parses a Verilog file using the slang library, identifies the specified cell, and extracts the logic expressions for its outputs. It returns a map where the keys are output signal names and the values are their corresponding logic expressions as strings.

Parameters

verilog_file	The path to the Verilog file to parse.
cell	The name of the cell (module) for which to extract logic expressions.

Returns

A map of output signal names to their logic expressions. Returns an empty map if parsing fails, the cell is not found, or no logic expressions can be derived.

Note

The function uses the slang library for Verilog parsing. Ensure that slang is properly installed and configured before using this function.

The logic extraction process involves traversing the syntax tree of the Verilog code and identifying relevant assignments and expressions within the specified cell.

Error messages and warnings are logged using the spdlog library.

Definition at line 755 of file LogicExtractor.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

7.19 LogicExtractor.hpp

Go to the documentation of this file.

```
00001 // include/LogicExtractor.hpp
00002
00003 #ifndef LOGIC_EXTRACTOR_HPP
00004 #define LOGIC_EXTRACTOR_HPP
00005
00006 #include "verilog_utils.hpp"
00007
00008 // Structure to represent a gate instance's information
00009 struct GateInfo {
00010   slang::parsing::TokenKind kind; // Store the TokenKind for reliable checks
00011   std::string gateTypeName;       // Store the string name like "and", "xor"
00012   std::vector<std::string> inputSignals;
```

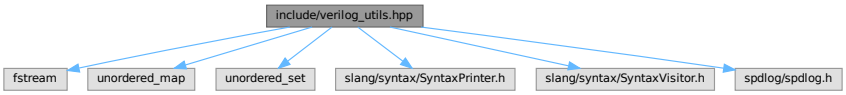
```
00013   std::string outputSignal;
00014 };
00015
00016 // Visitor class to extract netlist information and derive logic expressions
00017 class LogicExtractor : public slang::syntax::SyntaxVisitor<LogicExtractor> {
00018 public:
00019   // Constructor takes the target cell name
00020   explicit LogicExtractor(const std::string &targetCell)
00021       : targetCell_(targetCell), inTargetModule_(false), parsingComplete_(false) {}
00022
00023   // --- Visitor Handlers ---
00024   void handle(const slang::syntax::ModuleDeclarationSyntax &module);
00025   void handle(
00026       const slang::syntax::PortDeclarationSyntax &portDecl); // Handles direction/type declaration
00027   void
00028   handle(const slang::syntax::NonAnsiPortListSyntax &portList); // Handles port names list in header
00029   void
00030   handle(const slang::syntax::NetDeclarationSyntax &netDecl); // To find explicitly declared wires
00031   void handle(const slang::syntax::PrimitiveInstantiationSyntax
00032                   &primitiveInst); // Handles gate instantiation
00033   // Potentially handle ContinuousAssignSyntax if needed later:
00034   // void handle(const slang::syntax::ContinuousAssignSyntax& assign);
00035
00036   // --- Access Extracted Info (for Step 1 debugging) ---
00037   const std::unordered_map<std::string, GateInfo> &getExtractedGates()const {
00038     return gateOutputDrivers_;
00039   }
00040   const std::unordered_set<std::string> &getPrimaryInputs()const { return primaryInputs_; }
00041   const std::unordered_set<std::string> &getPrimaryOutputs()const { return primaryOutputs_; }
00042   const std::unordered_set<std::string> &getInternalWires()const { return internalWires_; }
00043
00044   // --- Logic Derivation (Commented out for Step 1) ---
00045   std::map<std::string, std::string> getLogicExpressions();
00046
00047 private:
00048   // --- Internal State ---
00049   const std::string &targetCell_;
00050   bool inTargetModule_;
00051   bool parsingComplete_; // Flag to indicate AST traversal is done
00052
00053   // Netlist Information
00054   std::unordered_set<std::string> primaryInputs_;
00055   std::unordered_set<std::string> primaryOutputs_;
00056   std::unordered_set<std::string> internalWires_; // Includes gate outputs
00057
00058   // Temporary map to store directions found in PortDeclarationSyntax
00059   // Key: port name, Value: direction ("input", "output", "inout", "unknown")
00060   std::unordered_map<std::string, std::string> portDirections_;
00061
00062   // Map: output signal name -> GateInfo driving it
00063   std::unordered_map<std::string, GateInfo> gateOutputDrivers_;
00064
00065   // Map: signal name -> Logic expression string (Memoization Cache) - (Used in Step 2)
00066   std::unordered_map<std::string, std::string> logicCache_;
00067
00068   // --- Helper Methods ---
00069   // Recursive function to derive logic for a given signal (Used in Step 2)
00070   std::string deriveLogicRecursive(const std::string &signalName);
00071   // Helper to format expressions based on gate type (Used in Step 2)
00072   std::string formatExpression(const GateInfo &gateInfo,
00073                                const std::vector<std::string> &inputExprs);
00074 };
00075
00076 // --- Function Declaration ---
00077 // For Step 1, this function will just run the visitor and maybe print summary
00078 void extractAndPrintNetlistInfo(const std::string &verilog_file, const std::string &cell);
00079
00080 // Function to be implemented in Step 2
00081 std::map<std::string, std::string> extractLogicFromVerilog(const std::string &verilog_file,
```

```
00082                                                   const std::string &cell);
00083
00084 #endif // LOGIC_EXTRACTOR_HPP
```

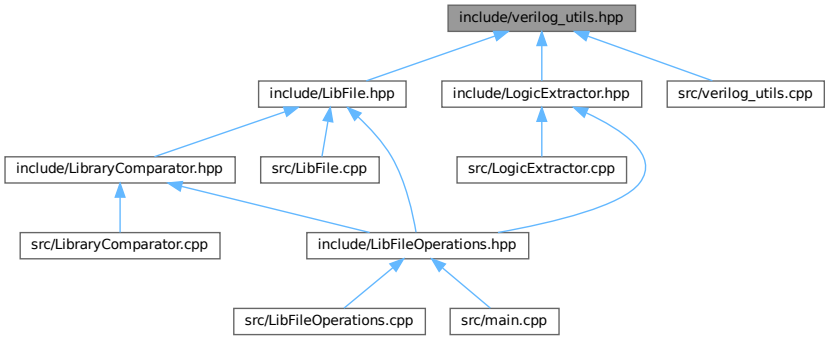
7.20 include/verilog_utils.hpp File Reference

```
#include <fstream>
#include <unordered_map>
#include <unordered_set>
#include "slang/syntax/SyntaxPrinter.h"
#include "slang/syntax/SyntaxVisitor.h"
#include "spdlog/spdlog.h"
```

Include dependency graph for verilog_utils.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class VerilogVisitor
- class CellExtractor
- class CellPrinter
- class ModuleRewriter

Functions

- void getAST (const std::string &verilog_file, const std::string &cell)

7.20.1 Function Documentation

7.20.1.1 getAST()

```
void getAST (
                const std::string & verilog_file,
                const std::string & cell )
```

Definition at line 544 of file verilog_utils.cpp.

Here is the call graph for this function:

7.21 verilog_utils.hpp

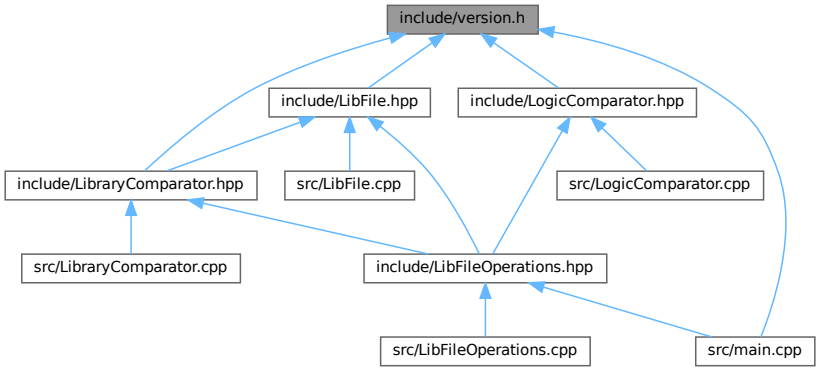
Go to the documentation of this file.

```
00001 // verilog_utils.hpp
00002
00003 #ifndef VERILOG_UTILS_H
00004 #define VERILOG_UTILS_H
00005
00006 #include <fstream>
00007 #include <unordered_map>
00008 #include <unordered_set>
00009
00010 #include "slang/syntax/SyntaxPrinter.h"
00011 #include "slang/syntax/SyntaxVisitor.h"
00012 #include "spdlog/spdlog.h" // Include spdlog
00013
00014 // Creating custom visitor class
00015 class VerilogVisitor : public slang::syntax::SyntaxVisitor<VerilogVisitor> {
00016 public:
00017   explicit VerilogVisitor(const std::string &targetCell)
00018       : targetCell_(targetCell), depth_(0), inTargetModule_(false) {}
00019   void handle(const slang::syntax::SyntaxNode &node);
00020   void handle(const slang::syntax::ModuleDeclarationSyntax &module);
```

```
00021   void handle(const slang::syntax::PortDeclarationSyntax &portDecl);
00022   void handle(const slang::syntax::HierarchyInstantiationSyntax &hierarchyInst);
00023   // void handle(const slang::syntax::PrimitiveInstantiationSyntax &primitiveInst);
00024   void handle(const slang::syntax::SpecifyBlockSyntax &specifyBlock);
00025
00026 private:
00027   const std::string &targetCell_;
00028   int depth_;
00029   bool inTargetModule_;
00030 };
00031
00032 // Creating a Rewriter to extract a specific cell
00033 class CellExtractor : public slang::syntax::SyntaxRewriter<CellExtractor> {
00034 public:
00035   explicit CellExtractor(const std::string &targetCell)
00036       : targetCell_(targetCell), foundTarget_(false) {}
00037   void handle(const slang::syntax::ModuleDeclarationSyntax &module);
00038   bool foundTargetCell() const;
00039
00040 private:
00041   const std::string &targetCell_;
00042   bool foundTarget_;
00043 };
00044
00045 // Print specific cell when visiting SyntaxTree
00046 class CellPrinter : public slang::syntax::SyntaxVisitor<CellPrinter> {
00047 public:
00048   explicit CellPrinter(const std::string &targetCell, std::ostream &out)
00049       : targetCell_(targetCell), out_(out), foundTarget_(false) {}
00050   void handle(const slang::syntax::ModuleDeclarationSyntax &module);
00051
00052 private:
00053   const std::string &targetCell_;
00054   std::ostream &out_;
00055   bool foundTarget_;
00056 };
00057
00058 // Comprehensive module rewriter for adding ports, instances, and connections
00059 class ModuleRewriter : public slang::syntax::SyntaxRewriter<ModuleRewriter> {
00060 public:
00061   explicit ModuleRewriter(const std::vector<std::string> &inputPins,
00062                           const std::vector<std::string> &outputPins,
00063                           const std::pair<std::string, std::string> &supercell_entry,
00064                           int instance_count, std::shared_ptr<spdlog::logger> logger)
00065       : inputPins_(inputPins), outputPins_(outputPins), cellName_(supercell_entry.first),
00066         moduleName_(supercell_entry.second), logger_(logger), depth_(0),
00067         instance_count_(instance_count) {}
00068   std::shared_ptr<spdlog::logger> logger_;
00069   void handle(const slang::syntax::SyntaxNode &node);
00070   void handle(const slang::syntax::ModuleDeclarationSyntax &module);
00071
00072 private:
00073   const std::vector<std::string> &inputPins_;
00074   const std::vector<std::string> &outputPins_;
00075   const std::string cellName_;
00076   const std::string moduleName_;
00077   std::map<std::string, std::string> portInfoMap_; // Map from port name to direction
00078   int depth_;
00079   int instance_count_;
00080 };
00081
00082 void getAST(const std::string &verilog_file, const std::string &cell);
00083
00084 #endif // VERILOG_UTILS_H
```

7.22 include/version.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define APP_NAME "ZlibValidation"
- #define APP_VERSION_MAJOR 1
- #define APP_VERSION_MINOR 1
- #define APP_VERSION_PATCH 2
- #define APP_VERSION "1.1.2"
- #define APP_AUTHOR "Song Zixuan"
- #define APP_CONTACT "cedar@zju.edu.cn"
- #define BUILD_TIMESTAMP "2025-04-15 16:33:51"

7.22.1 Macro Definition Documentation

7.22.1.1 APP_AUTHOR

```
#define APP_AUTHOR "Song Zixuan"
```

Definition at line 10 of file version.h.

7.22.1.2 APP_CONTACT

```
#define APP_CONTACT "cedar@zju.edu.cn"
```

Definition at line 11 of file version.h.

7.22.1.3 APP_NAME

```
#define APP_NAME "ZlibValidation"
```

Definition at line 5 of file version.h.

7.22.1.4 APP_VERSION

```
#define APP_VERSION "1.1.2"
```

Definition at line 9 of file version.h.

7.22.1.5 APP_VERSION_MAJOR

```
#define APP_VERSION_MAJOR 1
```

Definition at line 6 of file version.h.

7.22.1.6 APP_VERSION_MINOR

```
#define APP_VERSION_MINOR 1
```

Definition at line 7 of file version.h.

7.22.1.7 APP_VERSION_PATCH

```
#define APP_VERSION_PATCH 2
```

Definition at line 8 of file version.h.

7.22.1.8 BUILD_TIMESTAMP

```
#define BUILD_TIMESTAMP "2025-04-15 16:33:51"
```

Definition at line 12 of file version.h.

7.23 version.h

Go to the documentation of this file.

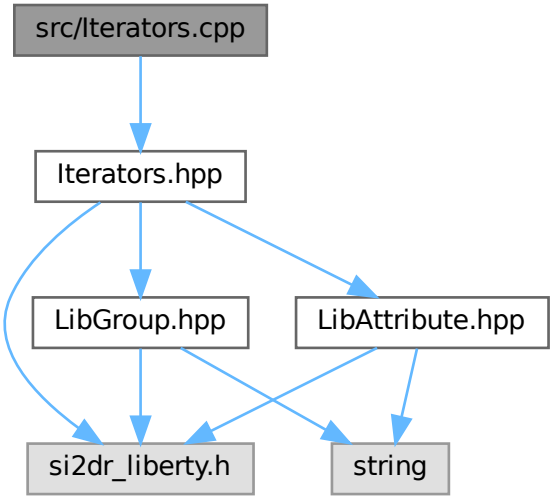
```
00001 // version.h.in
00002 #pragma once
00003
00004 // Auto-generated by CMake - DO NOT EDIT
00005 #define APP_NAME          "ZlibValidation"
00006 #define APP_VERSION_MAJOR 1
00007 #define APP_VERSION_MINOR 1
00008 #define APP_VERSION_PATCH 2
00009 #define APP_VERSION       "1.1.2"
00010 #define APP_AUTHOR        "Song Zixuan"
00011 #define APP_CONTACT       "cedar@zju.edu.cn"
00012 #define BUILD_TIMESTAMP   "2025-04-15 16:33:51"
```

7.24 README.md File Reference

7.25 src/Iterators.cpp File Reference

`#include "Iterators.hpp"`

Include dependency graph for Iterators.cpp:



7.26 Iterators.cpp

Go to the documentation of this file.

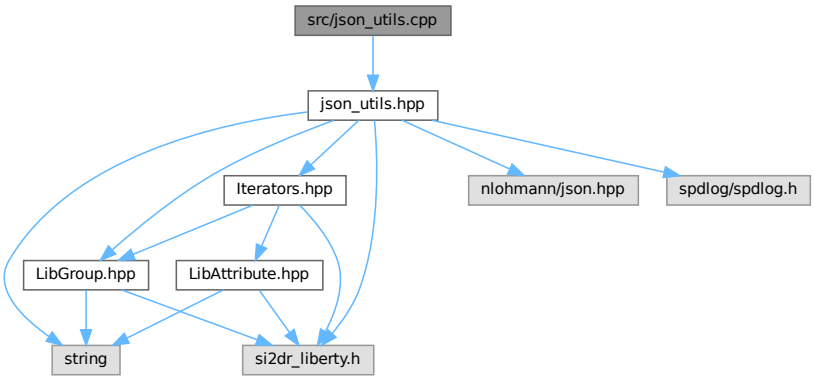
```
00001 #include "Iterators.hpp"
00002
00003 GroupsIterator::GroupsIterator(si2drGroupsIdT groups, si2drErrorT &err)
00004     : groups_(groups), err_(err) {
00005   group_ = si2drIterNextGroup(groups_, &err_);
00006 }
00007 GroupsIterator::~GroupsIterator() { si2drIterQuit(groups_, &err_); }
00008
00009 void GroupsIterator::next() { group_ = si2drIterNextGroup(groups_, &err_); }
00010 bool GroupsIterator::end() { return si2drObjectIsNull(group_, &err_); }
00011
00012 LibGroup GroupsIterator::get() { return LibGroup(group_, err_); }
00013
00014 AttributesIterator::AttributesIterator(si2drAttrsIdT attrs, si2drErrorT &err)
00015     : attrs_(attrs), err_(err) {
00016   attr_ = si2drIterNextAttr(attrs_, &err_);
00017 }
00018 AttributesIterator::~AttributesIterator() { si2drIterQuit(attrs_, &err_); }
00019
00020 void AttributesIterator::next() { attr_ = si2drIterNextAttr(attrs_, &err_); }
```

```
00021 bool AttributesIterator::end() { return si2drObjectIsNull(attr_, &err_); }
00022
00023 LibAttribute AttributesIterator::get() { return LibAttribute(attr_, err_); }
00024
00025 ValuesIterator::ValuesIterator(si2drValuesIdT values, si2drErrorT &err)
00026     : values_(values), err_(err) {
00027   si2drIterNextComplexValue(values_, &vtype_, &int_, &float_, &str_, &bool_, &exprp_, &err_);
00028 }
00029 ValuesIterator::~ValuesIterator() { si2drIterQuit(values_, &err_); }
00030
00031 void ValuesIterator::next() {
00032   si2drIterNextComplexValue(values_, &vtype_, &int_, &float_, &str_, &bool_, &exprp_, &err_);
00033 }
00034 bool ValuesIterator::end() { return vtype_ == SI2DR_UNDEFINED_VALUETYPE; }
```

7.27 src/json_utils.cpp File Reference

`#include "json_utils.hpp"`

Include dependency graph for json_utils.cpp:



Functions

- std::vector< double > parseStringToVector (const std::string &str)

  Parses a string representation of a vector of doubles into a vector of doubles.
- json generateLutJson (LibGroup &lib_lut_group, si2drErrorT &err)

  Generates a JSON object representation of a look-up table (LUT) from a LibGroup object.
- json generateTimingJson (LibGroup &lib_timing_group, si2drErrorT &err)

  Generates a JSON representation of timing information from a library timing group.
- json generatePowerJson (LibGroup &lib_power_group, si2drErrorT &err)

  Converts a Liberty power group into a JSON representation.
- std::pair< std::string, json > generatePinJson (LibGroup &lib_pin_group, si2drErrorT &err)

  Generates a JSON representation of a Liberty pin group.
- json generateCellJson (LibGroup &lib_cell_group, si2drErrorT &err)

  Generates a JSON representation of a cell from a LibGroup object.

7.27.1 Function Documentation

7.27.1.1 generateCellJson()

```
json generateCellJson (
                LibGroup & lib_cell_group,
                si2drErrorT & err )
```

Generates a JSON representation of a cell from a LibGroup object.

This function processes a LibGroup representing a cell and converts it into a JSON object. It extracts the cell name and processes specific attributes such as area, cell_leakage_power, and cell_footprint. It also processes pins within the cell by categorizing them based on their direction (input, output, internal, inout).

Parameters

lib_cell_group	The LibGroup object representing the cell
err	Reference to a si2drErrorT object for error handling

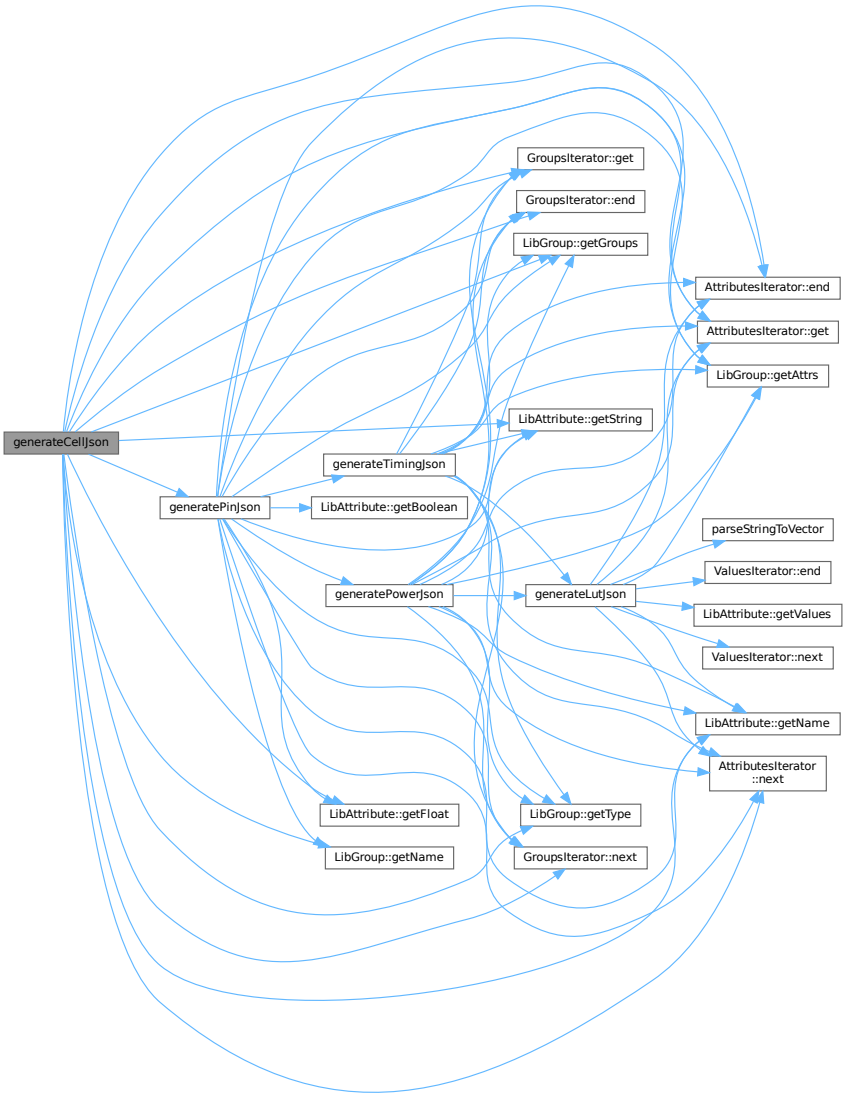
Returns

json A JSON object containing the cell's information with the following structure:

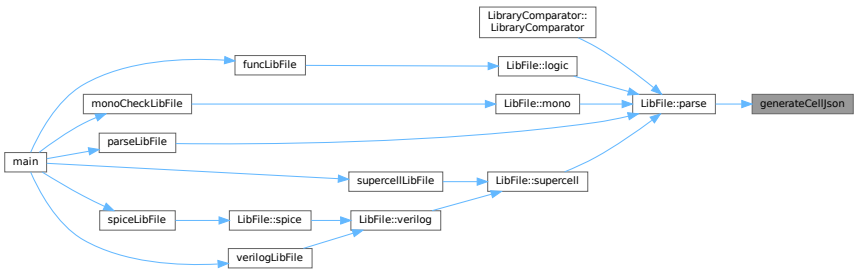
- "cell_name": string - name of the cell
- "area": float (optional) - cell area if present
- "cell_leakage_power": float (optional) - cell leakage power if present
- "cell_footprint": string (optional) - cell footprint if present
- "input_pins": array - list of input pins
- "output_pins": array - list of output pins
- "internal_pins": array - list of internal pins
- "inout_pins": array - list of inout pins

Definition at line 271 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.27.1.2 generateLutJson()

```
json generateLutJson (
                LibGroup & lib_lut_group,
                si2drErrorT & err )
```

Generates a JSON object representation of a look-up table (LUT) from a LibGroup object.

This function iterates through the attributes of the provided LibGroup object representing a LUT and converts them into a JSON structure. It specifically handles the following attributes:

- "index_1": Converted to a vector and stored in the JSON
- "index_2": Converted to a vector and stored in the JSON
- "values": Each value is parsed into a vector and added to an array in the JSON

Any other attributes encountered will trigger a warning message.

Parameters

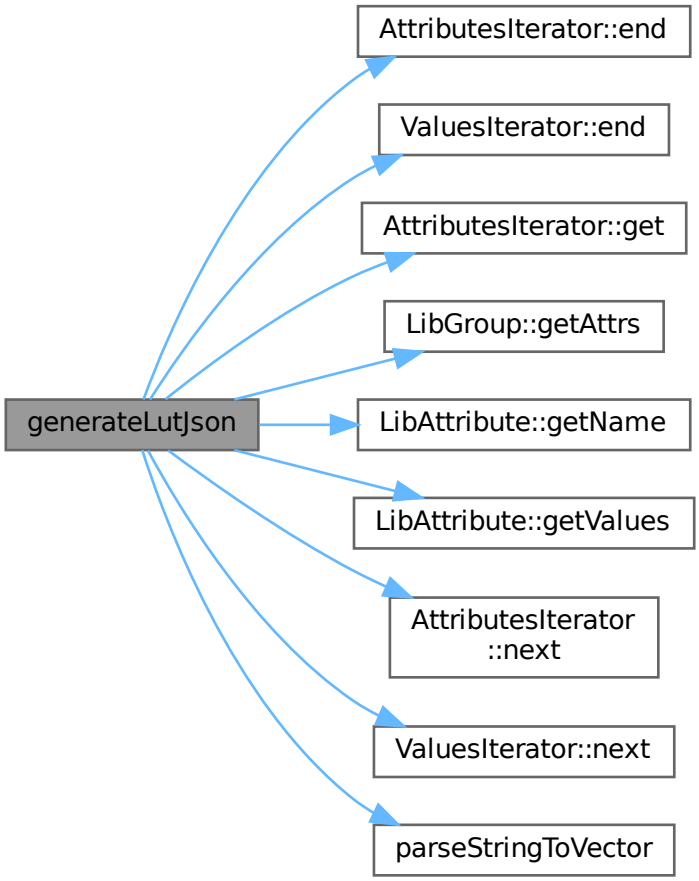
lib_lut_group	The LibGroup object containing the LUT data to be converted
err	Reference to an si2drErrorT object to track any errors during processing

Returns

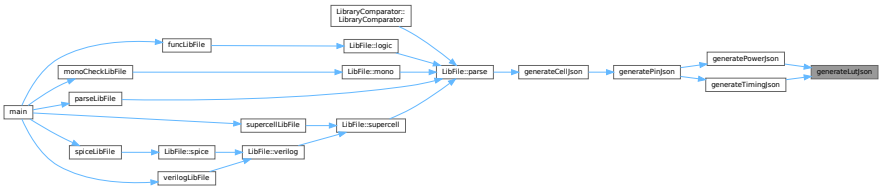
json A JSON object representing the LUT data with keys for "index_1", "index_2", and "values"

Definition at line 60 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.27.1.3 generatePinJson()

```
std::pair< std::string, json > generatePinJson (
```

```
LibGroup & lib_pin_group,
si2drErrorT & err )
```

Generates a JSON representation of a Liberty pin group.

This function traverses a Liberty pin group, extracts relevant attributes and sub-groups, and converts them into a JSON object. It also determines the pin's direction.

Processes the following pin attributes:

- direction
- max_transition, capacitance, rise_capacitance, fall_capacitance, max_capacitance (float types)
- function, power_down_function, related_ground_pin, related_power_pin, three_state (string types)
- clock (boolean type)

Handles the following sub-groups:

- internal_power: Converted using generatePowerJson()
- timing: Converted using generateTimingJson()

Parameters

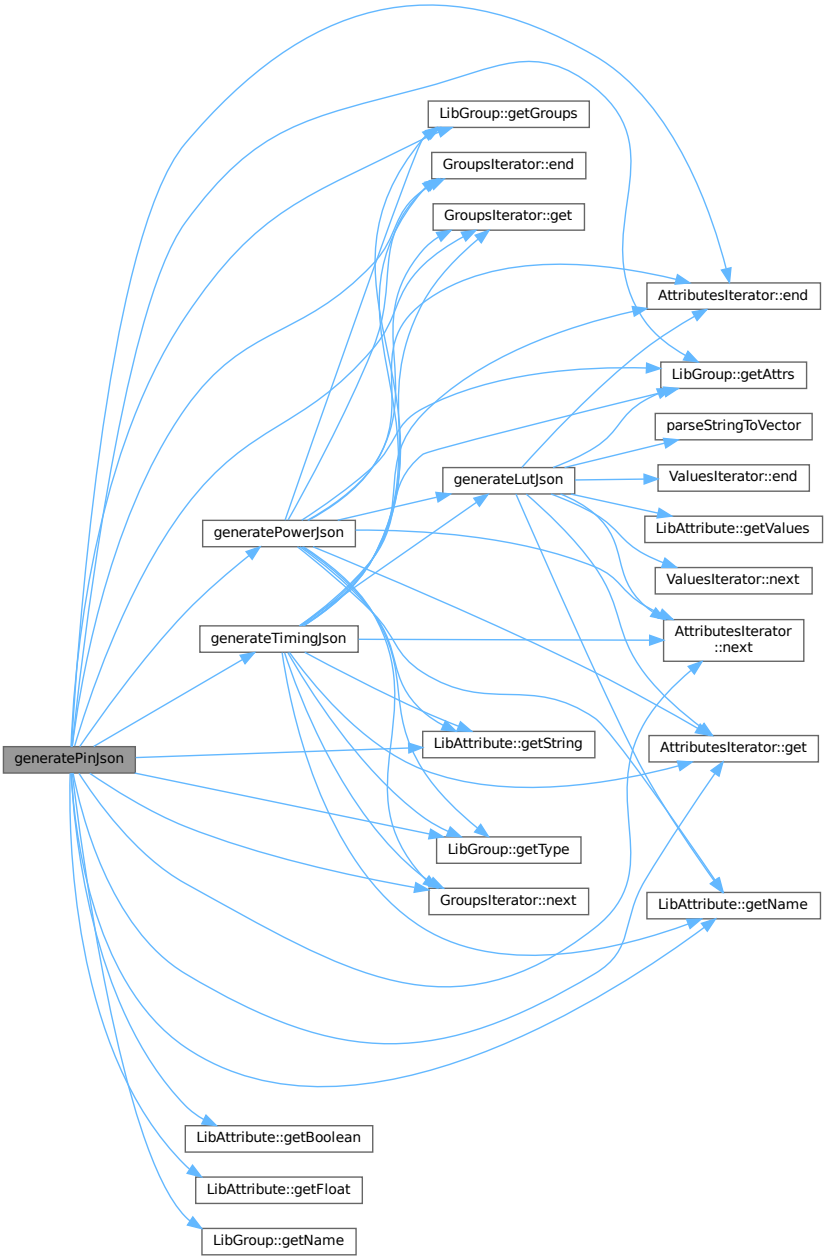
lib_pin_group	The Liberty pin group to process
err	Reference to an error object for tracking Liberty API errors

Returns

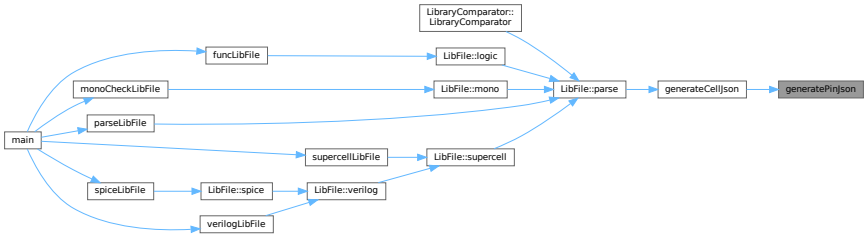
A pair containing the pin direction (string) and the JSON representation of the pin

Definition at line 205 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.27.1.4 generatePowerJson()

```
json generatePowerJson (
                LibGroup & lib_power_group,
                si2drErrorT & err )
```

Converts a Liberty power group into a JSON representation.

This function processes a Liberty power group and converts its attributes and subgroups into a JSON object. It handles attributes like "when", "related_pin", and "related_pg_pin", as well as "rise_power" and "fall_power" subgroups.

Parameters

lib_power_group	The Liberty power group to convert
err	Reference to an si2drErrorT object for error handling

Returns

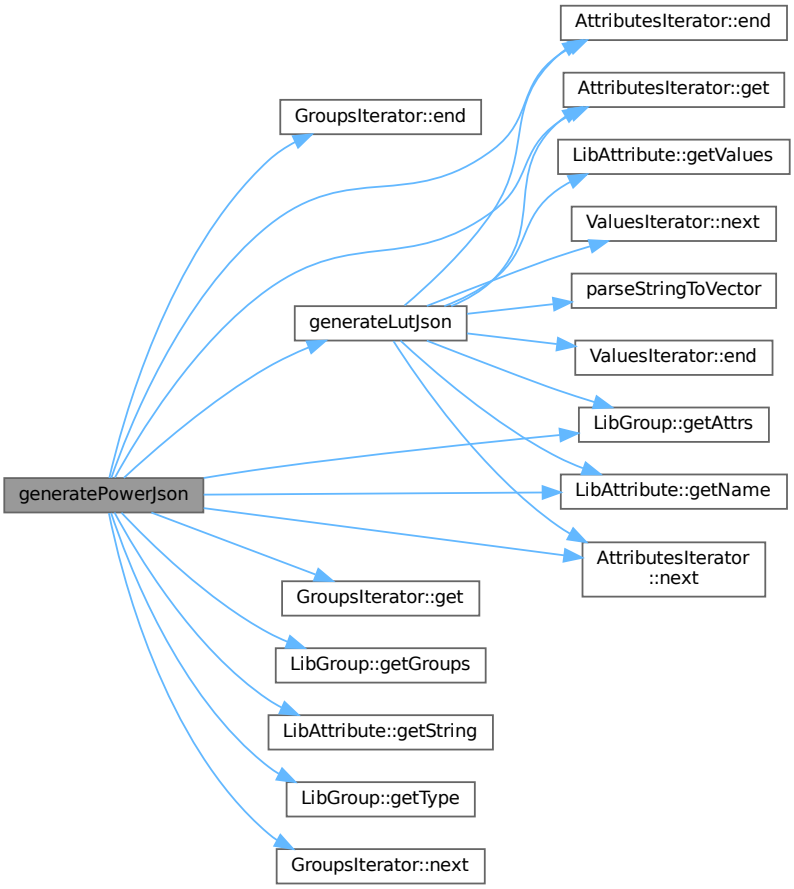
json A JSON object representing the power group data

The function:

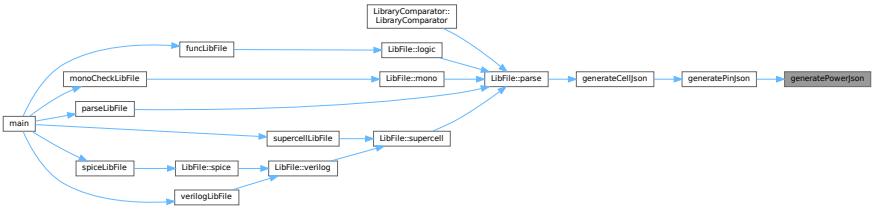
- Extracts string attributes (when, related_pin, related_pg_pin)
- Processes rise_power and fall_power subgroups by converting them to LUT JSON format
- Logs warnings for unknown power subgroup types

Definition at line 152 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.27.1.5 generateTimingJson()

```
json generateTimingJson (
                LibGroup & lib_timing_group,
                si2drErrorT & err )
```

Generates a JSON representation of timing information from a library timing group.

This function extracts timing attributes and sub-groups from a Liberty timing group and organizes them into a JSON object. It processes standard timing attributes like 'related_pin', 'timing_type', 'timing↩_sense', and 'when', as well as timing tables such as 'cell_fall', 'cell_rise', 'fall_transition', 'rise_↩transition', 'fall_constraint', and 'rise_constraint'.

Parameters

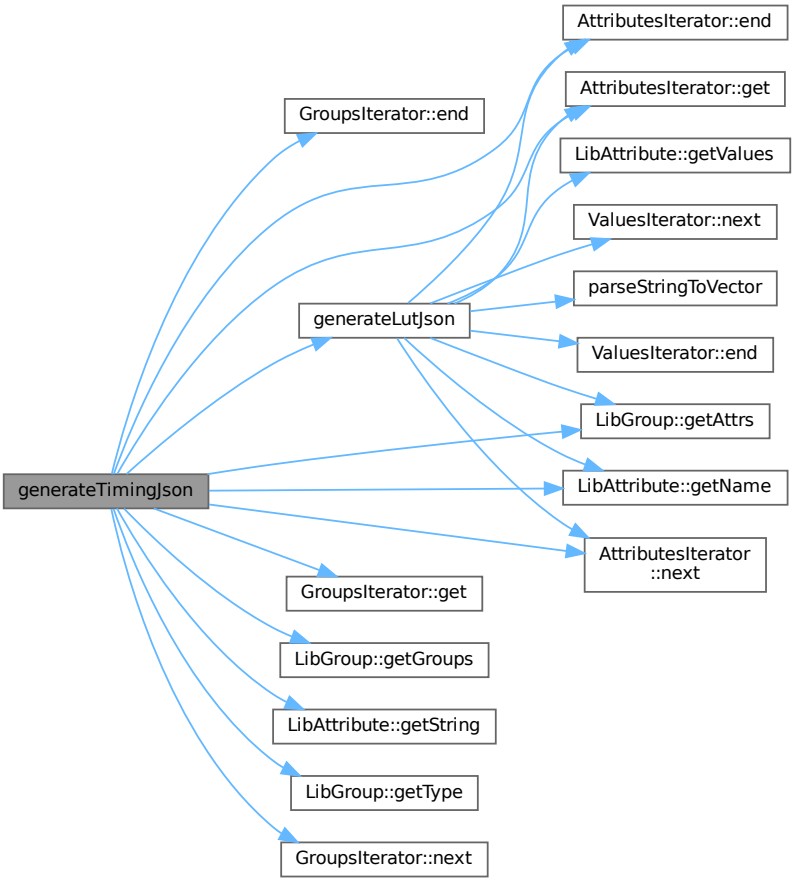
lib_timing_group	The Liberty timing group to process
err	Reference to an si2drErrorT object for error reporting

Returns

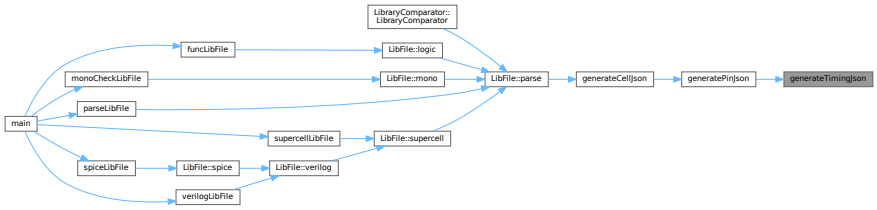
json A JSON object containing the extracted timing information

Definition at line 104 of file json_utils.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.27.1.6 parseStringToVector()

```
std::vector< double > parseStringToVector (
                const std::string & str )
```

Parses a string representation of a vector of doubles into a vector of doubles.

This function takes a string containing comma-separated numbers, cleans it by removing backslashes and newline characters, and then converts each comma-separated value into a double that is added to the resulting vector.

Parameters

str	The input string to be parsed, containing comma-separated numbers

Returns

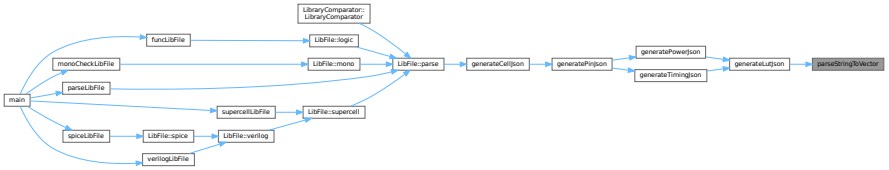
std::vector<double> A vector of doubles parsed from the input string

Exceptions

std::invalid_argument	If the string contains values that cannot be converted to double
std::out_of_range	If the string contains values that are out of range for double

Definition at line 25 of file json_utils.cpp.

Here is the caller graph for this function:



7.28 json_utils.cpp

Go to the documentation of this file.

```
00001 #include "json_utils.hpp"
00002
00003 /*
00004 json 类型的值可以是以下几种类型之一：
00005 方括号 []：用于包含一组有序的值（数组）。
00006 数组中的每个值可以是任意类型的 JSON 值，包括对象、数组、字符串、数字、布尔值或 null。
00007 花括号 {}：用于包含一组键值对（对象）。
00008 对象中的每个键都是一个字符串，值可以是任意类型的 JSON 值，包括对象、数组、字符串、数字、布尔值或
00009 null。
00010 */
00011
00025 std::vector<double> parseStringToVector(const std::string &str) {
00026   std::vector<double> result;
00027   std::string cleaned_str;
00028
00029   // Remove backslashes and newline characters
00030   for (char c : str) {
00031     if (c != '\\' && c != '\n') {
00032       cleaned_str += c;
00033     }
00034   }
00035
00036   std::stringstream ss(cleaned_str);
00037   std::string item;
00038   while (std::getline(ss, item, ',')) {
00039     result.push_back(std::stod(item));
00040   }
00041   return result;
00042 }
00043
00060 json generateLutJson(LibGroup &lib_lut_group, si2drErrorT &err) {
00061   json lut_json;
00062
00063   AttributesIterator attr_iter(lib_lut_group.getAttrs(), err);
00064   for (; !attr_iter.end(); attr_iter.next()) {
00065     LibAttribute lib_attr = attr_iter.get();
00066     std::string lut_attr_name = lib_attr.getName();
00067
00068     // spdlog::debug("LUT Attribute Name: {}", lib_attr.getName());
00069     // spdlog::debug("Is Complex? {}", lib_attr.isComplex());
00070
00071     if (lut_attr_name == "index_1" || lut_attr_name == "index_2") {
00072       ValuesIterator values_iter(lib_attr.getValues(), err);
00073       for (; !values_iter.end(); values_iter.next()) {
00074         // spdlog::debug("Type: {}", int(values_iter.vtype_)); // 5 is string
00075         // spdlog::debug("Str: {}", values_iter.str_);
00076         lut_json[lut_attr_name] = parseStringToVector(std::string(values_iter.str_));
00077       }
00078     } else if (lut_attr_name == "values") {
```

```
00079       ValuesIterator values_iter(lib_attr.getValues(), err);
00080       for (; !values_iter.end(); values_iter.next()) {
00081         // spdlog::debug("{}", values_iter.str_);
00082         lut_json["values"].push_back(parseStringToVector(std::string(values_iter.str_)));
00083       }
00084     } else {
00085       spdlog::warn("Unknown LUT attribute name: {}", lut_attr_name);
00086     }
00087   }
00088   return lut_json;
00089 }
00090
00104 json generateTimingJson(LibGroup &lib_timing_group, si2drErrorT &err) {
00105   json timing_json;
00106
00107   AttributesIterator attr_iter(lib_timing_group.getAttrs(), err);
00108   for (; !attr_iter.end(); attr_iter.next()) {
00109     LibAttribute lib_attr = attr_iter.get();
00110
00111     std::string attr_name = lib_attr.getName();
00112     if (attr_name == "related_pin" || attr_name == "timing_type" || attr_name == "timing_sense" ||
00113         attr_name == "when") {
00114       timing_json[attr_name] = lib_attr.getString();
00115     }
00116     // More timing attributes can be added here
00117   }
00118
00119   GroupsIterator timing_sub_group_iter(lib_timing_group.getGroups(), err);
00120   for (; !timing_sub_group_iter.end(); timing_sub_group_iter.next()) {
00121     LibGroup lib_timing_sub_group = timing_sub_group_iter.get();
00122
00123     std::string timing_sub_group_type = lib_timing_sub_group.getType();
00124     // std::string timing_sub_group_name = lib_timing_sub_group.getName();
00125     if (timing_sub_group_type == "cell_fall" || timing_sub_group_type == "cell_rise" ||
00126         timing_sub_group_type == "fall_transition" || timing_sub_group_type == "rise_transition" ||
00127         timing_sub_group_type == "fall_constraint" || timing_sub_group_type == "rise_constraint") {
00128       timing_json[timing_sub_group_type] = generateLutJson(lib_timing_sub_group, err);
00129     } else {
00130       spdlog::warn("Unknown timing sub group type: {}", timing_sub_group_type);
00131     }
00132   }
00133   return timing_json;
00134 }
00135
00152 json generatePowerJson(LibGroup &lib_power_group, si2drErrorT &err) {
00153   json power_json;
00154
00155   AttributesIterator attr_iter(lib_power_group.getAttrs(), err);
00156   for (; !attr_iter.end(); attr_iter.next()) {
00157     LibAttribute lib_attr = attr_iter.get();
00158
00159     std::string attr_name = lib_attr.getName();
00160     if (attr_name == "when" || attr_name == "related_pin" || attr_name == "related_pg_pin") {
00161       power_json[attr_name] = lib_attr.getString();
00162     }
00163     // More power attributes can be added here
00164   }
00165
00166   GroupsIterator power_sub_group_iter(lib_power_group.getGroups(), err);
00167   for (; !power_sub_group_iter.end(); power_sub_group_iter.next()) {
00168     LibGroup lib_power_sub_group = power_sub_group_iter.get();
00169
00170     std::string power_sub_group_type = lib_power_sub_group.getType();
00171     // std::string power_sub_group_name = lib_power_sub_group.getName();
00172     if (power_sub_group_type == "rise_power") {
00173       // spdlog::debug("Rise Power: {}", power_sub_group_name);
00174       power_json["rise_power"] = generateLutJson(lib_power_sub_group, err);
00175     } else if (power_sub_group_type == "fall_power") {
00176       power_json["fall_power"] = generateLutJson(lib_power_sub_group, err);
```

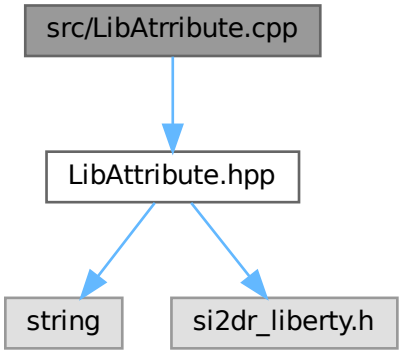
```
00177     } else {
00178       spdlog::warn("Unknown power sub group type: {}", power_sub_group_type);
00179     }
00180   }
00181   return power_json;
00182 }
00183
00205 std::pair<std::string, json> generatePinJson(LibGroup &lib_pin_group, si2drErrorT &err) {
00206   json pin_json;
00207   std::string direction;
00208   pin_json["pin_name"] = lib_pin_group.getName();
00209
00210   AttributesIterator attr_iter(lib_pin_group.getAttrs(), err);
00211   for (; !attr_iter.end(); attr_iter.next()) {
00212     LibAttribute lib_attr = attr_iter.get();
00213
00214     std::string attr_name = lib_attr.getName();
00215     if (attr_name == "direction") {
00216       direction = lib_attr.getString();
00217     } else if (attr_name == "max_transition" || attr_name == "capacitance" ||
00218                attr_name == "rise_capacitance" || attr_name == "fall_capacitance" ||
00219                attr_name == "max_capacitance") {
00220       pin_json[attr_name] = lib_attr.getFloat();
00221     } else if (attr_name == "function" || attr_name == "power_down_function" ||
00222                attr_name == "related_ground_pin" || attr_name == "related_power_pin" ||
00223                attr_name == "three_state") {
00224       pin_json[attr_name] = lib_attr.getString();
00225     } else if (attr_name == "clock") {
00226       pin_json[attr_name] = lib_attr.getBoolean();
00227     }
00228     // More pin attributes can be added here
00229   }
00230
00231   GroupsIterator pin_sub_group_iter(lib_pin_group.getGroups(), err);
00232   for (; !pin_sub_group_iter.end(); pin_sub_group_iter.next()) {
00233     LibGroup lib_pin_sub_group = pin_sub_group_iter.get();
00234
00235     std::string pin_sub_group_type = lib_pin_sub_group.getType();
00236     // std::string pin_sub_group_name = lib_pin_sub_group.getName();
00237     if (pin_sub_group_type == "internal_power") {
00238       json power_json = generatePowerJson(lib_pin_sub_group, err);
00239       pin_json["power_arcs"].push_back(power_json);
00240     } else if (pin_sub_group_type == "timing") {
00241       // spdlog::debug("Has Timing: {}", lib_pin_group.getName());
00242       json timing_json = generateTimingJson(lib_pin_sub_group, err);
00243       pin_json["timing_arcs"].push_back(timing_json);
00244     } else {
00245       spdlog::warn("Unknown pin sub group type: {}", pin_sub_group_type);
00246     }
00247   }
00248   return std::make_pair(direction, pin_json);
00249 }
00250
00271 json generateCellJson(LibGroup &lib_cell_group, si2drErrorT &err) {
00272   json cell_json;
00273   cell_json["cell_name"] = lib_cell_group.getName();
00274
00275   AttributesIterator attr_iter(lib_cell_group.getAttrs(), err);
00276   for (; !attr_iter.end(); attr_iter.next()) {
00277     LibAttribute lib_attr = attr_iter.get();
00278
00279     std::string attr_name = lib_attr.getName();
00280     if (attr_name == "area" || attr_name == "cell_leakage_power") {
00281       cell_json[attr_name] = lib_attr.getFloat();
00282     } else if (attr_name == "cell_footprint") {
00283       cell_json[attr_name] = lib_attr.getString();
00284     }
00285     // More cell attributes can be added here
00286   }
```

```
00287
00288   GroupsIterator cell_sub_group_iter(lib_cell_group.getGroups(), err);
00289   for (; !cell_sub_group_iter.end(); cell_sub_group_iter.next()) {
00290     LibGroup lib_cell_sub_group = cell_sub_group_iter.get();
00291
00292     std::string cell_sub_group_type = lib_cell_sub_group.getType();
00293     // std::string cell_sub_group_name = lib_cell_sub_group.getName();
00294
00295     // spdlog::debug("Cell Sub Group Type: {}", cell_sub_group_type);
00296     // spdlog::debug("Cell Sub Group Name: {}", cell_sub_group_name);
00297
00298     if (cell_sub_group_type == "pin") {
00299       auto [direction, pin_json] = generatePinJson(lib_cell_sub_group, err);
00300       if (direction == "input") {
00301         cell_json["input_pins"].push_back(pin_json);
00302       } else if (direction == "output") {
00303         cell_json["output_pins"].push_back(pin_json);
00304       } else if (direction == "internal") {
00305         cell_json["internal_pins"].push_back(pin_json);
00306       } else if (direction == "inout") {
00307         cell_json["inout_pins"].push_back(pin_json);
00308       } else {
00309         spdlog::warn("Unknown direction: {}", direction);
00310       }
00311     }
00312   }
00313   return cell_json;
00314 }
```

7.29 src/LibAtrribute.cpp File Reference

`#include "LibAttribute.hpp"`

Include dependency graph for LibAtrribute.cpp:



7.30 LibAtrribute.cpp

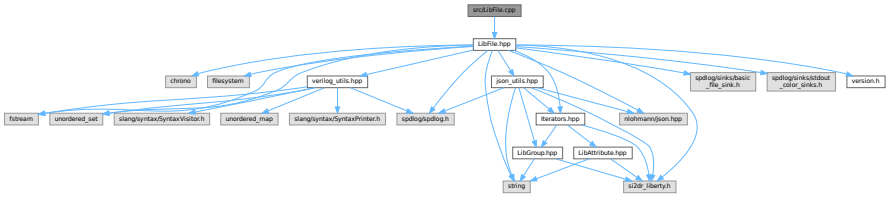
Go to the documentation of this file.

```
00001 #include "LibAttribute.hpp"
00002
00003 LibAttribute::LibAttribute(si2drAttrIdT attr, si2drErrorT &err) : attr_(attr), err_(err) {}
00004
00005 LibAttribute::~LibAttribute() {}
00006
00007 std::string LibAttribute::getName() {
00008   si2drStringT name = si2drAttrGetName(attr_, &err_);
00009   return name ? std::string(name) : std::string();
00010 }
00011
00012 bool LibAttribute::isComplex() { return si2drAttrGetAttrType(attr_, &err_) ? 1 : 0; }
00013
00014 si2drValuesIdT LibAttribute::getValues() { return si2drComplexAttrGetValues(attr_, &err_); }
00015
00016 long int LibAttribute::getInt() { return si2drSimpleAttrGetInt32Value(attr_, &err_); }
00017
00018 double LibAttribute::getFloat() { return si2drSimpleAttrGetFloat64Value(attr_, &err_); }
00019
00020 std::string LibAttribute::getString() {
00021   si2drStringT str = si2drSimpleAttrGetStringValue(attr_, &err_);
00022   return str ? std::string(str) : std::string();
00023 }
00024
00025 bool LibAttribute::getBoolean() { return si2drSimpleAttrGetBooleanValue(attr_, &err_) ? 1 : 0; }
```

7.31 src/LibFile.cpp File Reference

`#include "LibFile.hpp"`

Include dependency graph for LibFile.cpp:



7.32 LibFile.cpp

Go to the documentation of this file.

```
00001 #include "LibFile.hpp"
00002
00015 LibFile::LibFile(const std::string &filepath, const std::string &loggername)
00016     : filepath_(filepath), loggername_(loggername) {
00017   filename_ = filepath_.string();
00018   basename_ = filepath_.stem().string();
00019   jsonname_ = basename_ + ".json";
00020
00021   auto console_sink = std::make_shared<spdlog::sinks::stdout_color_sink_mt>();
00022   console_sink->set_level(spdlog::level::info);
00023
00024   auto file_sink = std::make_shared<spdlog::sinks::basic_file_sink_mt>(loggername_, true);
00025   file_sink->set_level(spdlog::level::debug);
```

```
00026
00027   std::vector<spdlog::sink_ptr> sinks{console_sink, file_sink};
00028   logger_ = std::make_shared<spdlog::logger>(loggername_, sinks.begin(), sinks.end());
00029   logger_->set_level(spdlog::level::debug);
00030
00031   logger_->info("Created LibFile object for '{}'", filename_);
00032   logger_->info("Debug log in: '{}'", loggername_);
00033 }
00034
00035 LibFile::~LibFile() { logger_->info("Closing file: '{}'", filename_); }
00036
00046 void LibFile::writeJsonToFile() {
00047   std::ofstream out(jsonname_);
00048   if (!out.is_open()) {
00049     logger_->error("Could not open file '{}' for writing", jsonname_);
00050     return;
00051   }
00052   out << lib_json_.dump(2);
00053   out.close();
00054   logger_->info("JSON data written to '{}'", jsonname_);
00055 }
00056
00077 void LibFile::read() {
00078   logger_->info("Reading '{}' ...", filename_);
00079
00080   auto start = std::chrono::high_resolution_clock::now();
00081   si2drReadLibertyFile(const_cast<char *>(filepath_.c_str()), &err_);
00082   auto end = std::chrono::high_resolution_clock::now();
00083   std::chrono::duration<double> duration = end - start;
00084
00085   if (err_ == SI2DR_INVALID_NAME) {
00086     logger_->error("Could not open file '{}' for parsing, quitting...", filename_);
00087     exit(301);
00088   } else if (err_ == SI2DR_SYNTAX_ERROR) {
00089     logger_->error("Syntax Errors were detected in the input file!");
00090     exit(401);
00091   } else {
00092     logger_->info("Done. Read time: {:.2f} seconds", duration.count());
00093   }
00094 }
00095
00116 void LibFile::parse() {
00117   si2drPIInit(&err_); // Initialize private error handler
00118   this->read();       // Read the Liberty file
00119
00120   logger_->info("Parsing '{}' ...", filename_);
00121   // Create a scope for the top-level groups iteration
00122   {
00123     GroupsIterator group_iter(si2drPIGetGroups(&err_), err_);
00124     for (; !group_iter.end(); group_iter.next()) {
00125       LibGroup lib_group = group_iter.get();
00126
00127       if (lib_group.getName() != "") {
00128         libname_ = lib_group.getName();
00129         lib_json_["library_name"] = this->libname_;
00130         logger_->info("Library Name: {}", libname_);
00131       } else {
00132         logger_->warn("Library Name: <NONAME>");
00133       }
00134       // Second level groups
00135       GroupsIterator sub_group_iter(lib_group.getGroups(), err_);
00136       for (; !sub_group_iter.end(); sub_group_iter.next()) {
00137         LibGroup lib_sub_group = sub_group_iter.get();
00138
00139         std::string sub_group_type = lib_sub_group.getType();
00140         std::string sub_group_name = lib_sub_group.getName();
00141
00142         if (sub_group_type == "cell") {
00143           logger_->debug("Cell Name: {}", sub_group_name);
```

```
00144           // if (sub_group_name != "AN2D0") {
00145           //   continue;
00146           // }
00147           // Handle each cell
00148           json cell_json = generateCellJson(lib_sub_group, err_);
00149           lib_json_["cells"].push_back(cell_json);
00150         } else if (sub_group_type == "operating_conditions") {
00151           logger_->info("Operating Conditions: {}", sub_group_name);
00152           // Get PVT from Operating Conditions
00153           AttributesIterator attr_iter(lib_sub_group.getAttrs(), err_);
00154           for (; !attr_iter.end(); attr_iter.next()) {
00155             LibAttribute lib_attr = attr_iter.get();
00156             logger_->debug("Attribute Name: {}", lib_attr.getName());
00157             logger_->debug("Int Value: {}", lib_attr.getInt());
00158             logger_->debug("Float Value: {}", lib_attr.getFloat());
00159             logger_->debug("String Value: {}", lib_attr.getString());
00160             if (lib_attr.getName() == "process") {
00161               process_ = lib_attr.getInt();
00162               lib_json_["process"] = process_;
00163             } else if (lib_attr.getName() == "voltage") {
00164               voltage_ = lib_attr.getFloat();
00165               // Round to 2 decimal places to avoid floating-point precision issues
00166               lib_json_["voltage"] = std::round(voltage_ * 100) / 100.0;
00167             } else if (lib_attr.getName() == "temperature") {
00168               temperature_ = lib_attr.getInt();
00169               lib_json_["temperature"] = temperature_;
00170             }
00171           }
00172           logger_->info("P: {}, V: {}, T: {}", process_, voltage_, temperature_);
00173         }
00174         // sub_group_iter's lifetime ends here, and the destructor is called
00175       }
00176     }
00177     // group_iter's lifetime ends here, and the destructor is called
00178   }
00179   si2drPIQuit(&err_);
00180 }
00181
00182 void LibFile::modify() { logger_->info("Modifying the file..."); }
00183
00205 bool LibFile::checkTimingArcMonotonicity(const json &cell, const json &pin, const json &arc,
00206                                          const std::string &timing_arc_name, const bool is_slew) {
00207   if (arc.contains("when") && !arc["when"].get<std::string>().empty()) {
00208     logger_->debug(
00209         "Checking cell: '{}', pin: '{}', related_pin: '{}', timing_arc: '{}', when: '{}'",
00210         cell["cell_name"].get<std::string>(), pin["pin_name"].get<std::string>(),
00211         arc["related_pin"].get<std::string>(), timing_arc_name, arc["when"].get<std::string>());
00212   } else {
00213     logger_->debug("Checking cell: '{}', pin: '{}', related_pin: '{}', timing_arc: '{}'",
00214                    cell["cell_name"].get<std::string>(), pin["pin_name"].get<std::string>(),
00215                    arc["related_pin"].get<std::string>(), timing_arc_name);
00216   }
00217   bool is_monotonic = true; // Assume the values are monotonic initially
00218   if (arc.contains(timing_arc_name) && arc[timing_arc_name].contains("values")) {
00219     std::vector<std::vector<double>> value_matrix;
00220     // Generate a matrix of values from the JSON array
00221     for (const auto &row_val : arc[timing_arc_name]["values"]) {
00222       std::vector<double> row_data;
00223       // Check if the value is an array
00224       if (!row_val.is_array()) {
00225         logger_->error("Invalid format: '{}' values should be an array of arrays in cell '{}', pin "
00226                        "'{}', related_pin '{}'",
00227                        timing_arc_name, cell["cell_name"].get<std::string>(),
00228                        pin["pin_name"].get<std::string>(), arc["related_pin"].get<std::string>());
00229         return false;
00230       }
00231       // Check for non-numeric values
00232       for (const auto &val : row_val) {
00233         if (val.is_number()) {
```

```
00234           row_data.push_back(val.get<double>());
00235         } else {
00236           logger_->warn("Non-numeric value found in '{}'.values, skipping value: {} in cell '{}', "
00237                         "pin '{}', related_pin '{}'",
00238                         timing_arc_name, val.dump(), cell["cell_name"].get<std::string>(),
00239                         pin["pin_name"].get<std::string>(), arc["related_pin"].get<std::string>());
00240         }
00241       }
00242       value_matrix.push_back(row_data);
00243     }
00244     // Check the matrix for monotonicity
00245     if (!value_matrix.empty() && !value_matrix[0].empty()) {
00246       // Check the monotonic incrementality of rows (by output load capacitance, index 2)
00247       for (size_t i = 0; i < value_matrix.size(); ++i) {
00248         for (size_t j = 1; j < value_matrix[i].size(); ++j) {
00249           if ((value_matrix[i][j] < value_matrix[i][j - 1] &&
00250                pin["pin_name"] != arc["related_pin"]) ||
00251               (value_matrix[i][j] == 0 && value_matrix[i][j - 1] == 0)) {
00252             // If contains "when" key, log the when value
00253             if (arc.contains("when") && !arc["when"].get<std::string>().empty()) {
00254               logger_->warn("Non-monotonic (by load) '{}' values: ({}, {}) {} < ({}, {}) {} "
00255                             "for Cell: {} Pin: {}->{} when: \"{}\"",
00256                             timing_arc_name, i, j, value_matrix[i][j], i, j - 1,
00257                             value_matrix[i][j - 1], cell["cell_name"].get<std::string>(),
00258                             arc["related_pin"].get<std::string>(),
00259                             pin["pin_name"].get<std::string>(), arc["when"].get<std::string>());
00260             } else {
00261               logger_->warn("Non-monotonic (by load) '{}' values: ({}, {}) {} < ({}, {}) {} "
00262                             "for Cell: {} Pin: {}->{}",
00263                             timing_arc_name, i, j, value_matrix[i][j], i, j - 1,
00264                             value_matrix[i][j - 1], cell["cell_name"].get<std::string>(),
00265                             arc["related_pin"].get<std::string>(),
00266                             pin["pin_name"].get<std::string>());
00267             }
00268             is_monotonic = false;
00269           }
00270         }
00271       }
00272       if (is_slew) {
00273         // Check the monotonic incrementality of columns (by input slew, index 1)
00274         for (size_t j = 0; j < value_matrix[0].size(); ++j) {
00275           for (size_t i = 1; i < value_matrix.size(); ++i) {
00276             if ((value_matrix[i][j] < value_matrix[i - 1][j] &&
00277                  pin["pin_name"] != arc["related_pin"]) ||
00278                 (value_matrix[i][j] == 0 && value_matrix[i - 1][j] == 0)) {
00279               // If contains "when" key, log the when value
00280               if (arc.contains("when") && !arc["when"].get<std::string>().empty()) {
00281                 logger_->warn("Non-monotonic (by slew) '{}' values: ({}, {}) {} < ({}, {}) {} "
00282                               "for Cell: {} Pin: {}->{} when: \"{}\"",
00283                               timing_arc_name, i, j, value_matrix[i][j], i - 1, j,
00284                               value_matrix[i - 1][j], cell["cell_name"].get<std::string>(),
00285                               arc["related_pin"].get<std::string>(),
00286                               pin["pin_name"].get<std::string>(), arc["when"].get<std::string>());
00287               } else {
00288                 logger_->warn("Non-monotonic (by slew) '{}' values: ({}, {}) {} < ({}, {}) {} "
00289                               "for Cell: {} Pin: {}->{}",
00290                               timing_arc_name, i, j, value_matrix[i][j], i - 1, j,
00291                               value_matrix[i - 1][j], cell["cell_name"].get<std::string>(),
00292                               arc["related_pin"].get<std::string>(),
00293                               pin["pin_name"].get<std::string>());
00294               }
00295               is_monotonic = false;
00296             }
00297           }
00298         }
00299       }
00300     } else {
00301       logger_->warn(
00302           "Empty or invalid 'values' array found for cell: '{}', pin: '{}', related_pin: '{}', "
```

```
00303           "timing_arc: '{}'",
00304           cell["cell_name"].get<std::string>(), pin["pin_name"].get<std::string>(),
00305           arc["related_pin"].get<std::string>(), timing_arc_name);
00306     }
00307   }
00308   return is_monotonic;
00309 }
00310
00331 void LibFile::mono(const bool is_slew) {
00332   /*
00333 * Use this command to check the following data in the current library to ensure
00334 * the tables are monotonically increasing with respect to output load:
00335 * cell_rise retaining_rise
00336 * cell_fall retaining_fall
00337 * rise_transition retain_rise_slew
00338 * fall_transition retain_fall_slew
00339 * mpw
00340 */
00341   logger_->info("Monotonicity check of '{}' starting ...", filename_);
00342
00343   if (!std::filesystem::exists(jsonname_)) {
00344     logger_->info("JSON file not found. Parsing Liberty file first.");
00345     this->parse();
00346     this->writeJsonToFile();
00347   } else {
00348     // Read the JSON file into json object
00349     std::ifstream in(jsonname_);
00350     if (!in.is_open()) {
00351       logger_->error("Could not open file '{}' for reading", jsonname_);
00352       return;
00353     }
00354     try {
00355       lib_json_ = json::parse(in);
00356     } catch (const json::parse_error &e) {
00357       logger_->error("JSON parsing error in file '{}': {}", jsonname_, e.what());
00358       in.close();
00359       return;
00360     }
00361     in.close();
00362   }
00363
00364   std::map<std::string, bool> cell_monotonicity_status; // Track pass/fail status for each cell
00365   std::vector<std::string> failed_cells;                // List of failed cell names
00366   int total_cells = 0;
00367   int passed_cells = 0;
00368
00369   // Check the monotonicity of delay values
00370   // The index 1 (time values) must be monotonically increasing and >= 0
00371   for (const auto &cell : lib_json_["cells"]) {
00372     total_cells++;
00373     bool cell_is_monotonic = true; // Assume cell is monotonic initially
00374     std::string cell_name = cell["cell_name"].get<std::string>();
00375     cell_monotonicity_status[cell_name] =
00376         true; // Initialize to pass, will be set to false if any check fails
00377
00378     if (cell.contains("output_pins")) {
00379       for (const auto &pin : cell["output_pins"]) {
00380         if (pin.contains("timing_arcs")) {
00381           for (const auto &arc : pin["timing_arcs"]) {
00382             // Check 4 timing arc names and accumulate monotonicity status
00383             for (const auto &name :
00384                  {"cell_rise", "cell_fall", "rise_transition", "fall_transition"}) {
00385               if (!checkTimingArcMonotonicity(cell, pin, arc, name, is_slew)) {
00386                 cell_is_monotonic = false;
00387               }
00388             }
00389           }
00390         }
00391       }
```

```
00392     }
00393     if (cell.contains("input_pins")) {
00394       for (const auto &pin : cell["input_pins"]) {
00395         if (!pin.contains("clock")) {
00396           continue; // Skip non-clock pins
00397         }
00398         // logger_->debug("Clock pin: {}", pin["pin_name"].get<std::string>());
00399         if (pin.contains("timing_arcs")) {
00400           for (const auto &arc : pin["timing_arcs"]) {
00401             if (arc.contains("timing_type")) {
00402               // logger_->debug("Timing type: {}", arc["timing_type"].get<std::string>());
00403               if (arc["timing_type"].get<std::string>() == "min_pulse_width") {
00404                 for (const auto &name : {"rise_constraint", "fall_constraint"}) {
00405                   if (!checkTimingArcMonotonicity(cell, pin, arc, name, is_slew)) {
00406                     cell_is_monotonic = false;
00407                   }
00408                 }
00409               }
00410             }
00411           }
00412         }
00413       }
00414     }
00415     // Update cell status based on all checks
00416     cell_monotonicity_status[cell_name] = cell_is_monotonic;
00417     if (cell_is_monotonic) {
00418       passed_cells++;
00419     } else {
00420       failed_cells.push_back(cell_name);
00421     }
00422   }
00423   logger_->info("Monotonicity check of '{}' completed.", filename_);
00424
00425   // Output summary statistics
00426   logger_->info("validate_monotonicity : {} out of {} cells passed", passed_cells, total_cells);
00427   logger_->info("validate_monotonicity : {} out of {} cells failed", total_cells - passed_cells,
00428                 total_cells);
00429   if (!failed_cells.empty()) {
00430     std::stringstream failed_cells_ss;
00431     for (size_t i = 0; i < failed_cells.size(); ++i) {
00432       failed_cells_ss << failed_cells[i];
00433       if (i < failed_cells.size() - 1) {
00434         failed_cells_ss << ", "; // Add comma if not the last element
00435       }
00436     }
00437     logger_->info("Failed cell list : {}", failed_cells_ss.str());
00438   }
00439 }
00440
00460 void LibFile::supercell(const int chain_length, const std::vector<std::string> &cell_names) {
00461   /*
00462 * Supercells are named as follows:
00463 *   <cellname>__X<chain_length>__<input_pin>__<output_pin>
00464 */
00465   logger_->info("Creating supercells for '{}'", filename_);
00466
00467   if (!std::filesystem::exists(jsonname_)) {
00468     logger_->info("JSON file not found. Parsing Liberty file first.");
00469     this->parse();
00470     this->writeJsonToFile();
00471   } else {
00472     // Read the JSON file into json object
00473     std::ifstream in(jsonname_);
00474     if (!in.is_open()) {
00475       logger_->error("Could not open file '{}' for reading", jsonname_);
00476       return;
00477     }
00478     try {
00479       lib_json_ = json::parse(in);
```

```
00480     } catch (const json::parse_error &e) {
00481       logger_->error("JSON parsing error in file '{}': {}", jsonname_, e.what());
00482       in.close();
00483       return;
00484     }
00485     in.close();
00486   }
00487
00488   // write to .map file
00489   std::ofstream out(basename_ + ".map");
00490   if (!out.is_open()) {
00491     logger_->error("Could not open file '{}' for writing", basename_ + ".map");
00492     return;
00493   }
00494   // if celll_names is empty, create supercells for all cells
00495   // else create supercells for only the specified cells
00496
00497   // Check if we're processing specific cells or all cells
00498   bool process_all_cells = cell_names.empty();
00499
00500   if (process_all_cells) {
00501     logger_->info("Creating supercells for ALL cells in '{}'", filename_);
00502   } else {
00503     logger_->info("Creating supercells for {} specified cells in '{}'", cell_names.size(),
00504                   filename_);
00505   }
00506
00507   // Create a set for faster lookups if we have specific cell names
00508   std::unordered_set<std::string> cell_set;
00509   std::unordered_set<std::string> found_cells;
00510   if (!process_all_cells) {
00511     cell_set.insert(cell_names.begin(), cell_names.end());
00512   }
00513
00514   for (const auto &cell : lib_json_["cells"]) {
00515     bool is_sequential = false;
00516     std::string cell_name = cell["cell_name"].get<std::string>();
00517
00518     // Skip cells not in the specified list
00519     if (!process_all_cells && cell_set.find(cell_name) == cell_set.end()) {
00520       continue;
00521     }
00522
00523     // Mark this cell as found
00524     if (!process_all_cells) {
00525       found_cells.insert(cell_name);
00526     }
00527
00528     logger_->debug("Creating supercells for cell: '{}'", cell_name);
00529
00530     std::vector<std::string> output_pins;
00531     std::vector<std::string> input_pins;
00532     // Extract input and output pins
00533     if (cell.contains("output_pins")) {
00534       for (const auto &pin : cell["output_pins"]) {
00535         output_pins.push_back(pin["pin_name"].get<std::string>());
00536       }
00537     }
00538     if (cell.contains("input_pins")) {
00539       for (const auto &pin : cell["input_pins"]) {
00540         if (pin.contains("clock")) {
00541           is_sequential = true;
00542           // clock pin not use for supercell
00543           continue;
00544         }
00545         input_pins.push_back(pin["pin_name"].get<std::string>());
00546       }
00547     }
00548
```

```
00549     // create supercells for all combinations of input and output pins
00550     for (const auto &output_pin : output_pins) {
00551       for (const auto &input_pin : input_pins) {
00552         if (is_sequential) {
00553           const int chain_len = 1;
00554           std::string supercell_name =
00555               cell_name + "__X" + std::to_string(chain_len) + "__" + input_pin + "__" + output_pin;
00556           out << cell_name << " " << supercell_name << std::endl;
00557         } else {
00558           std::string supercell_name = cell_name + "__X" + std::to_string(chain_length) + "__" +
00559                                        input_pin + "__" + output_pin;
00560           out << cell_name << " " << supercell_name << std::endl;
00561         }
00562       }
00563     }
00564   }
00565
00566   // Check for cells that were specified but not found
00567   if (!process_all_cells) {
00568     std::vector<std::string> not_found_cells;
00569     for (const auto &requested_cell : cell_names) {
00570       if (found_cells.find(requested_cell) == found_cells.end()) {
00571         not_found_cells.push_back(requested_cell);
00572       }
00573     }
00574
00575     // Output warning for cells that weren't found
00576     if (!not_found_cells.empty()) {
00577       std::string missing_cells = not_found_cells[0];
00578       for (size_t i = 1; i < not_found_cells.size(); ++i) {
00579         missing_cells += ", " + not_found_cells[i];
00580       }
00581       logger_->warn("{} specified cell{} not found in the library: {}", not_found_cells.size(),
00582                     not_found_cells.size() > 1 ? "s were" : " was", missing_cells);
00583     }
00584   }
00585
00586   out.close();
00587   logger_->info("Supercell creation complete in '{}'", basename_ + ".map");
00588 }
00589
00614 void LibFile::verilog(const int chain_length, const std::vector<std::string> &cell_names) {
00615   logger_->info("Creating Verilog for '{}'", filename_);
00616
00617   this->supercell(chain_length, cell_names);
00618
00619   // Create a temporary file for individual modules
00620   std::string temp_file = basename_ + "_temp.v";
00621   std::ofstream out(temp_file);
00622   if (!out.is_open()) {
00623     logger_->error("Could not open temp file '{}' for writing", temp_file);
00624     return;
00625   }
00626
00627   // Read the .map file into a vector to preserve all entries and their order
00628   std::vector<std::pair<std::string, std::string>> supercell_entries;
00629   std::ifstream in(basename_ + ".map");
00630   if (!in.is_open()) {
00631     logger_->error("Could not open file '{}' for reading", basename_ + ".map");
00632     return;
00633   }
00634
00635   std::string line;
00636   while (std::getline(in, line)) {
00637     std::istringstream iss(line);
00638     std::string cell_name, supercell_name;
00639     if (!(iss >> cell_name >> supercell_name)) {
00640       logger_->warn("Invalid line format in map file: '{}'", line);
00641       continue;
```

```
00642     }
00643     supercell_entries.push_back({cell_name, supercell_name});
00644   }
00645   in.close();
00646   logger_->info("Read {} supercells from '{}'", supercell_entries.size(), basename_ + ".map");
00647
00648   // Store module information for top-level creation
00649   std::vector<std::string> module_names;
00650   std::map<std::string, std::vector<std::string>> module_inputs;
00651   std::map<std::string, std::vector<std::string>> module_outputs;
00652
00653   // Generate verilog module for each supercell
00654   for (const auto &supercell_entry : supercell_entries) {
00655     // Check if the cell is sequential
00656     bool is_sequential = false;
00657     // Count the number of instances will be created in verilog
00658     int instance_count = 0;
00659     // Get original cell name from pair
00660     const std::string &cell_name = supercell_entry.first;
00661     // Get supercell name(as well as module name) from pair
00662     const std::string &module_name = supercell_entry.second;
00663
00664     // Store module name for top-level creation
00665     module_names.push_back(module_name);
00666
00667     logger_->info("Creating Module: '{}' from Cell '{}", module_name, cell_name);
00668
00669     // Get the cell JSON object
00670     json cell_json;
00671     for (const auto &cell : lib_json_["cells"]) {
00672       if (cell["cell_name"].get<std::string>() == cell_name) {
00673         cell_json = cell;
00674         break;
00675       }
00676     }
00677
00678     // Get input/output pins from the cell JSON object
00679     std::vector<std::string> input_pins;
00680     std::vector<std::string> output_pins;
00681     std::stringstream input_pins_ss;
00682     std::stringstream output_pins_ss;
00683     if (cell_json.contains("input_pins")) {
00684       for (const auto &pin : cell_json["input_pins"]) {
00685         if (pin.contains("clock")) {
00686           is_sequential = true;
00687         }
00688         std::string pin_name = pin["pin_name"].get<std::string>();
00689         input_pins.push_back(pin_name);
00690         input_pins_ss << pin_name << ", ";
00691
00692         // Store input pin name for top-level creation
00693         module_inputs[module_name].push_back(pin_name);
00694       }
00695     }
00696     if (cell_json.contains("output_pins")) {
00697       for (const auto &pin : cell_json["output_pins"]) {
00698         std::string pin_name = pin["pin_name"].get<std::string>();
00699         output_pins.push_back(pin_name);
00700         output_pins_ss << pin_name << ", ";
00701
00702         // Store output pin name for top-level creation
00703         module_outputs[module_name].push_back(pin_name);
00704       }
00705     }
00706     logger_->debug("Input pins set: {}", input_pins_ss.str());
00707     logger_->debug("Output pins set: {}", output_pins_ss.str());
00708
00709     // Sequential cells will have only one instance
00710     // Combinational cells will have instances equal to chain length
```

```
00711     if (is_sequential) {
00712       instance_count = 1;
00713     } else {
00714       instance_count = chain_length;
00715     }
00716 
00717     // Create ANSI port list
00718     std::string port_list_text = "(";
00719     // Add input ports
00720     bool isFirstPort = true;
00721     for (const auto &input_pin : input_pins) {
00722       if (!isFirstPort) {
00723         port_list_text += ", ";
00724       }
00725       port_list_text += "input " + input_pin;
00726       isFirstPort = false;
00727     }
00728     // Add output ports
00729     for (const auto &output_pin : output_pins) {
00730       if (!isFirstPort) {
00731         port_list_text += ", ";
00732       }
00733       port_list_text += "output " + output_pin;
00734       isFirstPort = false;
00735     }
00736     port_list_text += ")";
00737     logger_->debug("ANSI Port list: {}", port_list_text);
00738 
00739     // Creat full module header text
00740     std::string fullModuleText = "module " + module_name + port_list_text + ";\nendmodule";
00741     logger_->debug("Full module text: {}", fullModuleText);
00742 
00743     // Generate the syntax tree for the module using slang library
00744     auto tree = slang::syntax::SyntaxTree::fromText(fullModuleText);
00745     if (tree) {
00746       // Add ports to the syntax tree
00747       ModuleRewriter rewriter(input_pins, output_pins, supercell_entry, instance_count, logger_);
00748 
00749       tree = rewriter.transform(tree);
00750       // Revisit the syntax tree to check architecture
00751       // rewriter.visit(tree->root());
00752 
00753       // Output the transformed syntax tree
00754       out << "`timescale 1ns/10ps" << std::endl;
00755       out << slang::syntax::SyntaxPrinter::printFile(*tree) << std::endl << std::endl;
00756     } else {
00757       logger_->error("Failed to create syntax tree for module '{}'", module_name);
00758       continue;
00759     }
00760   }
00761 
00762   out.close();
00763 
00764   // Create the top-level module
00765   std::ofstream final_out(basename_ + ".v");
00766   if (!final_out.is_open()) {
00767     logger_->error("Could not open file '{}' for writing", basename_ + ".v");
00768     return;
00769   }
00770 
00771   // Collect all port names separately for inputs and outputs
00772   std::vector<std::string> all_input_ports;
00773   std::vector<std::string> all_output_ports;
00774   std::stringstream top_instances;
00775 
00776   // First pass: collect all port names
00777   for (const auto &module_name : module_names) {
00778     // Collect input port names
00779     for (const auto &pin : module_inputs[module_name]) {
```

```
00780       all_input_ports.push_back(module_name + "__" + pin);
00781     }
00782
00783     // Collect output port names
00784     for (const auto &pin : module_outputs[module_name]) {
00785       all_output_ports.push_back(module_name + "__" + pin);
00786     }
00787
00788     // Create instance
00789     std::string instance_name = "I_" + module_name.substr(0, module_name.find("__X"));
00790     for (size_t i = module_name.find("__X") + 3; i < module_name.length(); i++) {
00791       if (module_name[i] == '_' && module_name[i + 1] == '_') {
00792         instance_name += "__" + module_name.substr(i + 2);
00793         break;
00794       }
00795     }
00796
00797     // Generate port connections for this instance
00798     std::stringstream instance_ports;
00799     for (const auto &pin : module_inputs[module_name]) {
00800       instance_ports << "." << pin << "(" << module_name << "__" << pin << "), ";
00801     }
00802     for (const auto &pin : module_outputs[module_name]) {
00803       instance_ports << "." << pin << "(" << module_name << "__" << pin << "), ";
00804     }
00805
00806     // Remove last comma and space
00807     std::string instance_ports_str = instance_ports.str();
00808     if (!instance_ports_str.empty()) {
00809       instance_ports_str = instance_ports_str.substr(0, instance_ports_str.length() - 2);
00810     }
00811
00812     top_instances << "  " << module_name << "  " << instance_name << " (" << instance_ports_str
00813                   << ");\n";
00814   }
00815
00816   // Now generate the port list with all inputs first, then all outputs
00817   std::stringstream top_ports;
00818
00819   // Add all input ports first
00820   for (size_t i = 0; i < all_input_ports.size(); ++i) {
00821     top_ports << all_input_ports[i];
00822     if (i < all_input_ports.size() - 1 || !all_output_ports.empty()) {
00823       top_ports << ", ";
00824     }
00825   }
00826
00827   // Then add all output ports
00828   for (size_t i = 0; i < all_output_ports.size(); ++i) {
00829     top_ports << all_output_ports[i];
00830     if (i < all_output_ports.size() - 1) {
00831       top_ports << ", ";
00832     }
00833   }
00834
00835   // Generate input and output declarations
00836   std::stringstream top_inputs;
00837   std::stringstream top_outputs;
00838
00839   // Generate input declarations
00840   for (const auto &port : all_input_ports) {
00841     top_inputs << "  input " << port << ";\n";
00842   }
00843
00844   // Generate output declarations
00845   for (const auto &port : all_output_ports) {
00846     top_outputs << "  output " << port << ";\n";
00847   }
00848
```

```
00849   // Write the top module
00850   final_out << "`timescale 1ns/10ps" << std::endl;
00851   final_out << "module validate_top ( " << top_ports.str() << " );" << std::endl;
00852   final_out << std::endl; // Add newline before port declarations
00853   final_out << top_inputs.str();
00854   final_out << top_outputs.str();
00855   final_out << std::endl;
00856   final_out << top_instances.str();
00857   final_out << "endmodule" << std::endl << std::endl;
00858
00859   // Append the module definitions from the temporary file
00860   std::ifstream temp_in(temp_file);
00861   if (temp_in.is_open()) {
00862     final_out << temp_in.rdbuf();
00863     temp_in.close();
00864   } else {
00865     logger_->error("Could not open temp file '{}' for reading", temp_file);
00866   }
00867
00868   final_out.close();
00869
00870   // Remove temporary file
00871   // if (std::filesystem::exists(temp_file)) {
00872   //   std::filesystem::remove(temp_file);
00873   // }
00874
00875   logger_->info("Verilog creation complete in '{}'", basename_ + ".v");
00876 }
00877
00889 std::vector<std::string> LibFile::splitString(const std::string &s) {
00890   std::vector<std::string> tokens;
00891   std::stringstream ss(s);
00892   std::string token;
00893   while (ss >> token) {
00894     tokens.push_back(token);
00895   }
00896   return tokens;
00897 }
00898
00931 void LibFile::generateRCLines(std::ofstream &outFile, const std::string &netName, int instanceIndex,
00932                               bool isFinalStage) {
00933   // --- Default RC Values (as observed in the target SPICE) ---
00934   const double R1_INTERMEDIATE = 0.01;
00935   const double C1_INTERMEDIATE = 5.3e-16;
00936   const double R2_INTERMEDIATE = 0.01;
00937   const double R1_FINAL = 1e-2;
00938   const double C1_FINAL = 1e-18;
00939   const std::string CAP_GROUND = "COREGND1"; // Assumed ground for capacitors
00940
00941   std::string r1Name = "R1_" + std::to_string(instanceIndex);
00942   std::string c1Name = "C1_" + std::to_string(instanceIndex);
00943   std::string r2Name = "R2_" + std::to_string(instanceIndex);
00944   std::string node1 = netName + ":1"; // Intermediate node 1
00945   std::string node2 = netName + ":2"; // Intermediate node 2
00946
00947   // Set precision for scientific notation output
00948   outFile << std::scientific << std::setprecision(1);
00949
00950   if (!isFinalStage) {
00951     // Intermediate RCR structure (R1-C1-R2)
00952     outFile << r1Name << " " << node1 << " " << node2 << " " << R1_INTERMEDIATE << std::endl;
00953     outFile << c1Name << " " << node2 << " " << CAP_GROUND << " " << C1_INTERMEDIATE << std::endl;
00954     outFile << r2Name << " " << node2 << " " << netName << " " << R2_INTERMEDIATE << std::endl;
00955   } else {
00956     // Final RC structure (R1-C1 connected to the instance output node :1)
00957     // Note: The target SPICE connects R1 between node1 and the final port (netName).
00958     outFile << r1Name << " " << node1 << " " << netName << " " << R1_FINAL << std::endl;
00959     outFile << c1Name << " " << node1 << " " << CAP_GROUND << " " << C1_FINAL << std::endl;
00960   }
```

```
00961   // Reset precision/format to default
00962   outFile << std::defaultfloat << std::setprecision(6);
00963 }
00964
01013 bool LibFile::modifySpiceNetlist(const std::string &v2lvsSpiceFile, // Input is the v2lvs output
01014                                   const std::string &finalSpiceFile, // Output is the final file
01015                                   const std::string &targetGlobalLine) {
01016   logger_->info("Post-processing SPICE netlist: {} -> {}", v2lvsSpiceFile, finalSpiceFile);
01017
01018   std::ifstream inFile(v2lvsSpiceFile);
01019   // Write directly to the final output file, overwriting if it exists
01020   std::ofstream outFile(finalSpiceFile, std::ios::trunc);
01021
01022   if (!inFile) {
01023     logger_->error("Could not open input v2lvs SPICE file for modification: {}", v2lvsSpiceFile);
01024     return false;
01025   }
01026   if (!outFile) {
01027     logger_->error("Could not open final output SPICE file for writing: {}", finalSpiceFile);
01028     inFile.close(); // Close input file if output fails
01029     return false;
01030   }
01031
01032   // --- Add Metadata Comments ---
01033   outFile << "** Generated by " << APP_NAME << " v" << APP_VERSION << " from " << APP_AUTHOR;
01034   auto now = std::chrono::system_clock::now();
01035   std::time_t now_time_t = std::chrono::system_clock::to_time_t(now);
01036   // Use localtime_s on Windows, localtime_r on POSIX, or std::localtime (less safe)
01037   // std::tm now_tm;
01038   // localtime_s(&now_tm, &now_time_t); // Example for Windows
01039   std::tm *now_tm_ptr = std::localtime(&now_time_t); // Standard C++, potentially less thread-safe
01040   if (now_tm_ptr) {
01041     outFile << ". On: " << std::put_time(now_tm_ptr, "%c %Z") << " **\n" << std::endl;
01042   } else {
01043     outFile << ". Timestamp unavailable **\n" << std::endl;
01044   }
01045   // --- End Metadata ---
01046
01047   std::string line;
01048   bool includeFound = false;
01049   bool globalInserted = false;
01050   bool inSubckt = false;
01051   std::string previousInstanceLine = "";
01052   int instanceIndex = 0; // Counter for R/C naming within a subckt
01053
01054   while (std::getline(inFile, line)) {
01055     // Trim leading/trailing whitespace
01056     line.erase(0, line.find_first_not_of(" \t\n\r\f\v"));
01057     line.erase(line.find_last_not_of(" \t\n\r\f\v") + 1);
01058
01059     // Skip empty lines and v2lvs header comments
01060     if (line.empty() || line.find("$ Spice netlist generated by v2lvs") == 0 ||
01061         line.find("$ v2") == 0 || // Catch version line too
01062         line.find("*.BUSDELIMITER") == 0) {
01063       continue; // Skip these lines entirely
01064     }
01065
01066     // Write other comments directly after processing any pending instance
01067     if (line[0] == '*') {
01068       // Process previous instance before writing comment if needed
01069       if (!previousInstanceLine.empty()) {
01070         std::vector<std::string> tokens = this->splitString(previousInstanceLine);
01071         if (tokens.size() > 2 && (tokens[0][0] == 'X' || tokens[0][0] == 'x')) {
01072           std::string moduleName = tokens.back();
01073           std::string outputNet = tokens[tokens.size() - 2]; // Assumption
01074           bool isFinal = (outputNet == "CO" || outputNet == "S" || outputNet == "ZN");
01075           tokens[tokens.size() - 2] = outputNet + ":1"; // Modify output pin
01076
01077           // Reconstruct line with correct VDD/VSS order
```

```
01078           for (size_t i = 0; i < tokens.size() - 1; ++i) { // Up to module name
01079             outFile << tokens[i] << " ";
01080           }
01081           outFile << "VDD VSS " << moduleName << std::endl; // Insert VDD VSS before module name
01082
01083           this->generateRCLines(outFile, outputNet, instanceIndex - 1, isFinal);
01084         } else {
01085           logger_->warn("Previous line stored but not recognized as instance: {}",
01086                         previousInstanceLine);
01087           outFile << previousInstanceLine << std::endl;
01088         }
01089         previousInstanceLine = ""; // Clear after processing
01090       }
01091       outFile << line << std::endl; // Write the comment line
01092       continue;
01093     }
01094
01095     // Handle .INCLUDE
01096     if (line.find(".INCLUDE") == 0) {
01097       outFile << line << std::endl;
01098       includeFound = true;
01099       continue; // Process next line to insert global
01100     }
01101
01102     // Insert the target global line after .INCLUDE
01103     if (includeFound && !globalInserted) {
01104       outFile << targetGlobalLine << std::endl << std::endl; // Add extra newline for spacing
01105       globalInserted = true;
01106       // Fall through to process the current line
01107     }
01108
01109     // Handle .SUBCKT
01110     if (line.find(".SUBCKT") == 0 || line.find(".subckt") == 0) { // Handle case-insensitivity
01111       if (!previousInstanceLine.empty()) {
01112         std::vector<std::string> tokens = this->splitString(previousInstanceLine);
01113         if (tokens.size() > 2 && (tokens[0][0] == 'X' || tokens[0][0] == 'x')) {
01114           std::string moduleName = tokens.back();
01115           std::string outputNet = tokens[tokens.size() - 2];
01116           bool isFinal = (outputNet == "CO" || outputNet == "S" || outputNet == "ZN");
01117           tokens[tokens.size() - 2] = outputNet + ":1";
01118
01119           for (size_t i = 0; i < tokens.size() - 1; ++i) {
01120             outFile << tokens[i] << " ";
01121           }
01122           outFile << "VDD VSS " << moduleName << std::endl;
01123
01124           this->generateRCLines(outFile, outputNet, instanceIndex - 1, isFinal);
01125         } else {
01126           logger_->warn("Previous line stored but not recognized as instance: {}",
01127                         previousInstanceLine);
01128           outFile << previousInstanceLine << std::endl;
01129         }
01130         previousInstanceLine = "";
01131       }
01132
01133       inSubckt = true;
01134       instanceIndex = 0;                          // Reset R/C counter for new subcircuit
01135       outFile << line << " VDD VSS" << std::endl; // Add VDD VSS to ports
01136     }
01137     // Handle .ENDS
01138     else if (line.find(".ENDS") == 0 || line.find(".ends") == 0) { // Handle case-insensitivity
01139       inSubckt = false;
01140       // Process the last instance line *before* writing .ENDS
01141       if (!previousInstanceLine.empty()) {
01142         std::vector<std::string> tokens = this->splitString(previousInstanceLine);
01143         if (tokens.size() > 2 && (tokens[0][0] == 'X' || tokens[0][0] == 'x')) {
01144           std::string moduleName = tokens.back();
01145           std::string outputNet = tokens[tokens.size() - 2];
01146           bool isFinal = (outputNet == "CO" || outputNet == "S" || outputNet == "ZN");
```

```
01147           tokens[tokens.size() - 2] = outputNet + ":1";
01148
01149           for (size_t i = 0; i < tokens.size() - 1; ++i) {
01150             outFile << tokens[i] << " ";
01151           }
01152           outFile << "VDD VSS " << moduleName << std::endl;
01153
01154           this->generateRCLines(outFile, outputNet, instanceIndex - 1, isFinal);
01155         } else {
01156           logger_->warn("Previous line stored but not recognized as instance: {}",
01157                         previousInstanceLine);
01158           outFile << previousInstanceLine << std::endl;
01159         }
01160         previousInstanceLine = ""; // Clear after processing
01161       }
01162       outFile << line << std::endl << std::endl; // Add extra newline after .ENDS
01163     }
01164     // Skip original .GLOBAL lines
01165     else if (line.find(".GLOBAL") == 0 || line.find(".global") == 0) {
01166       continue;
01167     }
01168     // Handle Instance lines
01169     else if (inSubckt && (line[0] == 'X' || line[0] == 'x')) {
01170       if (!previousInstanceLine.empty()) {
01171         std::vector<std::string> tokens = this->splitString(previousInstanceLine);
01172         if (tokens.size() > 2 && (tokens[0][0] == 'X' || tokens[0][0] == 'x')) {
01173           std::string moduleName = tokens.back();
01174           std::string outputNet = tokens[tokens.size() - 2]; // Assumption
01175           bool isFinal = (outputNet == "CO" || outputNet == "S" || outputNet == "ZN");
01176           tokens[tokens.size() - 2] = outputNet + ":1"; // Modify output pin
01177
01178           // Reconstruct line with correct VDD/VSS order
01179           for (size_t i = 0; i < tokens.size() - 1; ++i) { // Up to module name
01180             outFile << tokens[i] << " ";
01181           }
01182           outFile << "VDD VSS " << moduleName << std::endl; // Insert VDD VSS before module name
01183
01184           this->generateRCLines(outFile, outputNet, instanceIndex - 1,
01185                                 isFinal); // Use previous index
01186         } else {
01187           logger_->warn("Previous line stored but not recognized as instance: {}",
01188                         previousInstanceLine);
01189           outFile << previousInstanceLine << std::endl;
01190         }
01191       }
01192       previousInstanceLine = line;
01193       instanceIndex++;
01194     }
01195     // Handle other lines
01196     else {
01197       if (!previousInstanceLine.empty()) {
01198         std::vector<std::string> tokens = this->splitString(previousInstanceLine);
01199         if (tokens.size() > 2 && (tokens[0][0] == 'X' || tokens[0][0] == 'x')) {
01200           std::string moduleName = tokens.back();
01201           std::string outputNet = tokens[tokens.size() - 2];
01202           bool isFinal = (outputNet == "CO" || outputNet == "S" || outputNet == "ZN");
01203           tokens[tokens.size() - 2] = outputNet + ":1";
01204
01205           for (size_t i = 0; i < tokens.size() - 1; ++i) {
01206             outFile << tokens[i] << " ";
01207           }
01208           outFile << "VDD VSS " << moduleName << std::endl;
01209
01210           this->generateRCLines(outFile, outputNet, instanceIndex - 1, isFinal);
01211         } else {
01212           logger_->warn("Previous line stored but not recognized as instance: {}",
01213                         previousInstanceLine);
01214           outFile << previousInstanceLine << std::endl;
01215         }
```

```
01216         previousInstanceLine = ""; // Clear after processing
01217       }
01218       outFile << line << std::endl;
01219     }
01220   }
01221
01222   // Process the very last instance line
01223   if (!previousInstanceLine.empty()) {
01224     std::vector<std::string> tokens = this->splitString(previousInstanceLine);
01225     if (tokens.size() > 2 && (tokens[0][0] == 'X' || tokens[0][0] == 'x')) {
01226       std::string moduleName = tokens.back();
01227       std::string outputNet = tokens[tokens.size() - 2];
01228       bool isFinal = (outputNet == "CO" || outputNet == "S" || outputNet == "ZN");
01229       tokens[tokens.size() - 2] = outputNet + ":1";
01230
01231       for (size_t i = 0; i < tokens.size() - 1; ++i) {
01232         outFile << tokens[i] << " ";
01233       }
01234       outFile << "VDD VSS " << moduleName << std::endl;
01235
01236       this->generateRCLines(outFile, outputNet, instanceIndex - 1, isFinal);
01237     } else {
01238       logger_->warn("Last stored line not recognized as instance: {}", previousInstanceLine);
01239       outFile << previousInstanceLine << std::endl;
01240     }
01241   }
01242
01243   inFile.close();
01244   outFile.close();
01245
01246   // No renaming needed, we wrote directly to the final file
01247   logger_->info("SPICE netlist post-processing successful for: {}", finalSpiceFile);
01248   return true;
01249 }
01250
01279 void LibFile::spice(const int chain_length, const std::vector<std::string> &cell_names,
01280                     const std::string &verilog_lib_file, const std::string &spice_lib_file) {
01281   logger_->info("Creating SPICE for '{}'", filename_);
01282
01283   // 1. Generate the temporary Verilog file first
01284   this->verilog(chain_length, cell_names);
01285
01286   // Define input and output filenames
01287   std::string temp_verilog_file = basename_ + "_temp.v";
01288   std::string v2lvs_output_spice_file = basename_ + ".v2lvs.spi"; // v2lvs output
01289   std::string final_output_spice_file = basename_ + ".spi";       // Final processed output
01290
01291   // Check if temporary Verilog file exists
01292   std::ifstream check_v_in(temp_verilog_file);
01293   if (!check_v_in.is_open()) {
01294     logger_->error("Temporary Verilog file {} not found or could not be opened. Verilog generation "
01295                    "might have failed.",
01296                    temp_verilog_file);
01297     return;
01298   }
01299   check_v_in.close(); // Close after checking
01300
01301   // 2. Check if V2LVS tool is available
01302   logger_->debug("Checking for v2lvs tool...");
01303   std::string v2lvs_path_cmd = "which v2lvs";
01304   std::string v2lvs_path_result;
01305   std::array<char, 128> buffer;
01306   std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(v2lvs_path_cmd.c_str(), "r"), pclose);
01307   if (!pipe) {
01308     logger_->error("Failed to run command to find v2lvs: {}", v2lvs_path_cmd);
01309     // Do NOT delete temp_verilog_file here if popen fails
01310     return;
01311   }
01312   while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
```

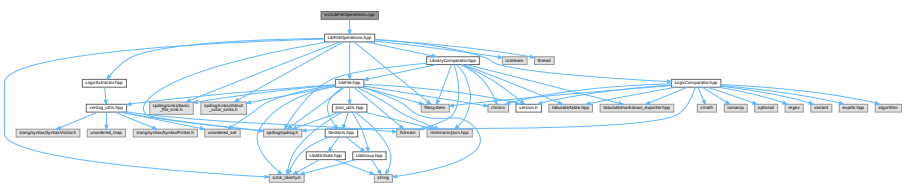
```
01313     v2lvs_path_result += buffer.data();
01314   }
01315
01316   // Trim potential newline from the result
01317   if (!v2lvs_path_result.empty() && v2lvs_path_result.back() == '\n') {
01318     v2lvs_path_result.pop_back();
01319   }
01320
01321   if (v2lvs_path_result.empty()) {
01322     logger_->error(
01323         "V2LVS tool not found in PATH. Please install Calibre or ensure v2lvs is accessible.");
01324     // Do NOT delete temp_verilog_file
01325     return;
01326   } else {
01327     logger_->info("V2LVS tool found at: {}", v2lvs_path_result);
01328   }
01329
01330   // 3. Construct and Run V2LVS command
01331   logger_->info("Running V2LVS to generate initial SPICE netlist...");
01332   std::string v2lvs_cmd_full = "v2lvs";               // Start with the command name
01333   v2lvs_cmd_full += " -v " + temp_verilog_file;       // Input Verilog
01334   v2lvs_cmd_full += " -o " + v2lvs_output_spice_file; // Output SPICE
01335   v2lvs_cmd_full += " -s0 VSS";                       // Default ground net, generates .GLOBAL VSS
01336   v2lvs_cmd_full += " -s1 VDD";                       // Default power net, generates .GLOBAL VDD
01337   v2lvs_cmd_full += " -i";                            // Use positional pins (traditional SPICE)
01338   v2lvs_cmd_full += " -l " + verilog_lib_file;        // Verilog library for pin order
01339   v2lvs_cmd_full += " -s " + spice_lib_file;          // SPICE library to include
01340
01341   logger_->debug("Executing V2LVS command: {}", v2lvs_cmd_full);
01342   int ret = system(v2lvs_cmd_full.c_str());
01343
01344   // Do NOT delete temp_verilog_file
01345
01346   if (ret != 0) {
01347     logger_->error("V2LVS command failed with exit code: {}. Command: {}", ret, v2lvs_cmd_full);
01348     // Attempt to remove potentially incomplete v2lvs output SPICE file
01349     std::remove(v2lvs_output_spice_file.c_str());
01350     return;
01351   }
01352   logger_->info("V2LVS completed successfully, output at '{}'.", v2lvs_output_spice_file);
01353
01354   // 4. Post-process the generated SPICE file
01355   std::string target_global = ".global VSS GND COREGND1 VDD COREVDD1";
01356
01357   // Call the modification function, reading from .v2lvs.spi and writing to .spi
01358   if (!this->modifySpiceNetlist(v2lvs_output_spice_file, final_output_spice_file, target_global)) {
01359     logger_->error("Failed to post-process the generated SPICE netlist: {}",
01360                    v2lvs_output_spice_file);
01361     // Keep both files for debugging if modification fails
01362     return;
01363   }
01364
01365   // 5. Final success message
01366   logger_->info("Intermediate v2lvs output kept in '{}'", v2lvs_output_spice_file);
01367   logger_->info("Temporary Verilog input kept in '{}'", temp_verilog_file);
01368   logger_->info("SPICE generation and post-processing complete. Final output in '{}'",
01369                 final_output_spice_file);
01370 }
01371
01392 std::map<std::string, std::string> LibFile::logic(const std::string &cell_name) {
01393   std::map<std::string, std::string> logic_map = {};
01394
01395   logger_->info("Getting output pin logic function for '{}'", filename_);
01396
01397   if (!std::filesystem::exists(jsonname_)) {
01398     logger_->info("JSON file not found. Parsing Liberty file first.");
01399     this->parse();
01400     this->writeJsonToFile();
01401   } else {
```

```
01402     // Read the JSON file into json object
01403     std::ifstream in(jsonname_);
01404     if (!in.is_open()) {
01405       logger_->error("Could not open file '{}' for reading", jsonname_);
01406       return logic_map;
01407     }
01408     try {
01409       lib_json_ = json::parse(in);
01410     } catch (const json::parse_error &e) {
01411       logger_->error("JSON parsing error in file '{}': {}", jsonname_, e.what());
01412       in.close();
01413       return logic_map;
01414     }
01415     in.close();
01416   }
01417
01418   // Check if the cell name exists in the JSON object
01419   for (const auto &cell : lib_json_["cells"]) {
01420     if (cell["cell_name"].get<std::string>() == cell_name) {
01421       logger_->debug("Found cell: '{}'", cell_name);
01422
01423       // Tranverse all output pins
01424       if (cell.contains("output_pins")) {
01425         for (const auto &pin : cell["output_pins"]) {
01426           std::string pin_name = pin["pin_name"].get<std::string>();
01427           // Check if the pin has a "timing_arcs" attribute
01428           if (pin.contains("function")) {
01429             std::string function = pin["function"].get<std::string>();
01430             logic_map[pin_name] = function;
01431             logger_->debug("Pin: '{}' Logic Function: '{}'", pin_name, function);
01432           } else {
01433             logger_->warn("Pin: '{}' does not have a 'function' attribute", pin_name);
01434           }
01435         }
01436       }
01437     }
01438   }
01439   if (logic_map.empty()) {
01440     logger_->warn("No logic functions found for cell: '{}'", cell_name);
01441   } else {
01442     logger_->info("Found {} Logic functions for cell: '{}'", logic_map.size(), cell_name);
01443     // Print the logic functions
01444     for (const auto &pair : logic_map) {
01445       logger_->info("Pin -> Logic Function: {} -> {}", pair.first, pair.second);
01446     }
01447   }
01448   return logic_map;
01449 }
```

7.33 src/LibFileOperations.cpp File Reference

`#include "LibFileOperations.hpp"`

Include dependency graph for LibFileOperations.cpp:



Functions

- void printInfo ()

  Sets up and configures the global logger and prints application information.
- void parseLibFile (const std::string &library_path, const std::string log_file_name)

  Parses a library file and generates corresponding output.
- void monoCheckLibFile (const std::string &library_path, const std::string log_file_name, bool is_slew)

  Performs monotonicity check on a library file.
- void supercellLibFile (const std::string &library_path, const std::string &log_file_name, int chain_length, const std::vector< std::string > &cell_names)

  Creates supercell map structures from a Liberty library file.
- void verilogLibFile (const std::string &library_path, const std::string &log_file_name, int chain_length, const std::vector< std::string > &cell_names)

  Generates Verilog files from a library file for specified cells.
- void spiceLibFile (const std::string &library_path, const std::string &log_file_name, int chain_length, const std::vector< std::string > &cell_names, const std::string &verilog_lib_file, const std::string &spice_lib_file)

  Generates a SPICE library file from a given library file, applying a specified chain length and cell names.
- void compareLibFiles (const std::string &ref_lib, const std::string &comp_lib, const double reltol, const double abstol, std::string &report_file_name)

  Compares two library files and generates a detailed comparison report.
- void funcLibFile (const std::string &ref_file, const std::string &comp_file, const std::vector< std::string > &cell_names, std::string &report_file_name)

  Performs a functional equivalence check between two files (Liberty or Verilog) for a given set of cells.

7.33.1 Function Documentation

7.33.1.1 compareLibFiles()

```
void compareLibFiles (
                const std::string & ref_lib,
                const std::string & comp_lib,
                const double reltol,
                const double abstol,
                std::string & report_file_name )
```

Compares two library files and generates a detailed comparison report.

This function compares a reference library file with another library file, performing validation checks based on specified tolerance parameters. The comparison results are written to a report file in markdown or text format.

Parameters

ref_lib	Path to the reference library file to use as the baseline
comp_lib	Path to the library file to compare against the reference
reltol	Relative tolerance for numerical comparisons (must be >= 0.0)
abstol	Absolute tolerance for numerical comparisons
report_file_name	[in,out] Name of the file to write the comparison report to. If empty, defaults to [comp_lib_basename].cmp.md. If provided but doesn't end with .txt or .md, .md will be appended.

Note

The function will log an error and return without comparing if reltol is invalid.

Log files will be created with the naming pattern [library_basename].cmp.log

The function uses the LibraryComparator class to perform the actual comparison.

Definition at line 249 of file LibFileOperations.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

7.33.1.2 funcLibFile()

```
void funcLibFile (
                const std::string & ref_file,
                const std::string & comp_file,
```

```
const std::vector< std::string > & cell_names,
std::string & report_file_name )
```

Performs a functional equivalence check between two files (Liberty or Verilog) for a given set of cells.

This function compares the logic functions of specified cells in two files, which can be either in Liberty (.lib) or Verilog (.v) format. It extracts the logic functions for each cell from both files, compares them, and generates a report summarizing the comparison results.

Parameters

ref_file	The path to the reference file (Liberty or Verilog).
comp_file	The path to the comparison file (Liberty or Verilog).
cell_names	A vector of cell names to be checked for functional equivalence.
report_file_name	A string to store the name of the report file. If empty, a default name is generated. If the provided name does not end with ".txt" or ".md", ".md" is appended.

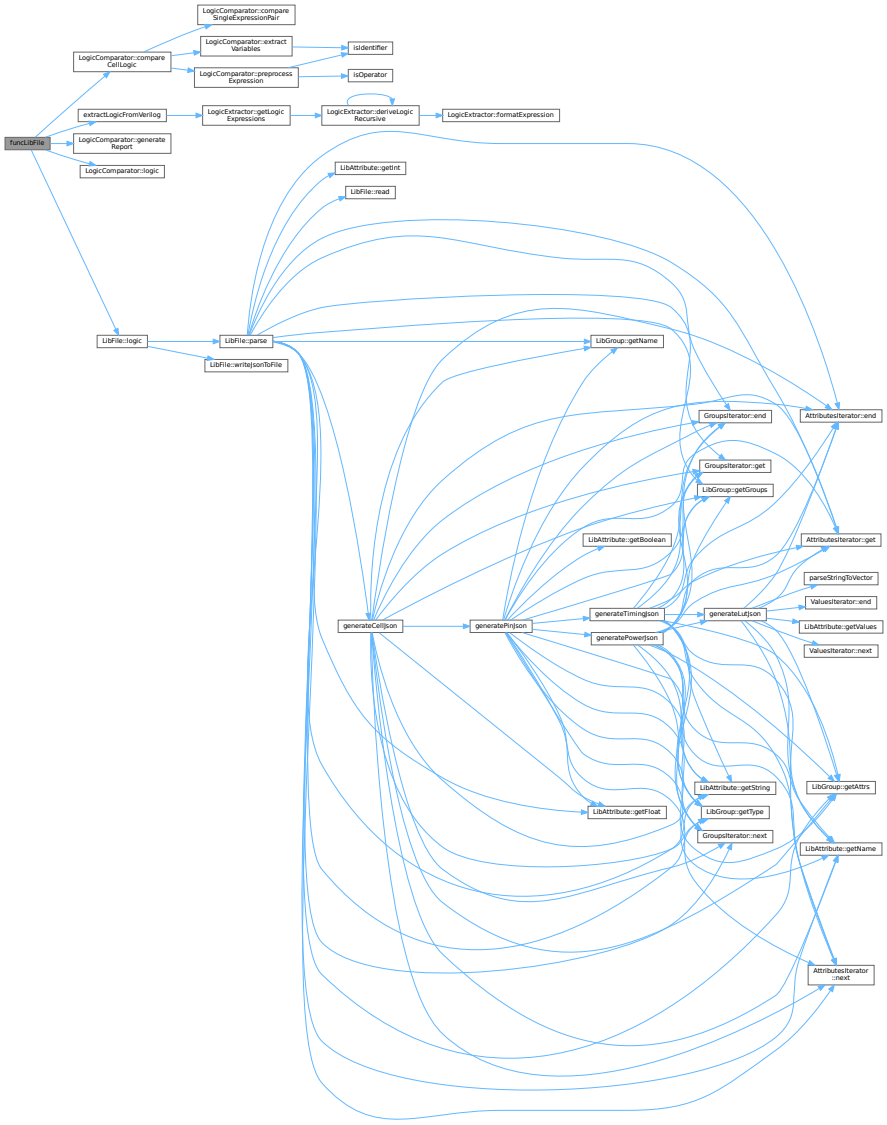
The function first checks the file extensions to determine the file format. It then extracts the logic functions for each specified cell from both files. The logic functions are then compared, and a report is generated, which includes the comparison results for each cell. The report is written to the specified report file.

Note

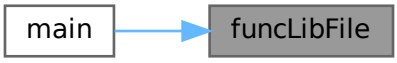
- If no cell names are provided, the function logs an error and returns.
- If the reference or comparison file format is not supported (i.e., not .lib or .v), the function logs an error and returns.
- The report file is cleared before writing the comparison results.
- The function uses spdlog for logging information, warnings, and errors.
- The function utilizes the LogicComparator class to perform the logic comparison and generate the report.
- Memory allocated for LibFile objects is managed using raw pointers and must be manually deallocated to prevent memory leaks.

Definition at line 308 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.33.1.3 monoCheckLibFile()

```
void monoCheckLibFile (
                const std::string & library_path,
                const std::string log_file_name,
                bool is_slew )
```

Performs monotonicity check on a library file.

This function validates the monotonicity of timing data in a library file. It creates a log file to record the results of the check and handles any exceptions that occur during the process.

Parameters

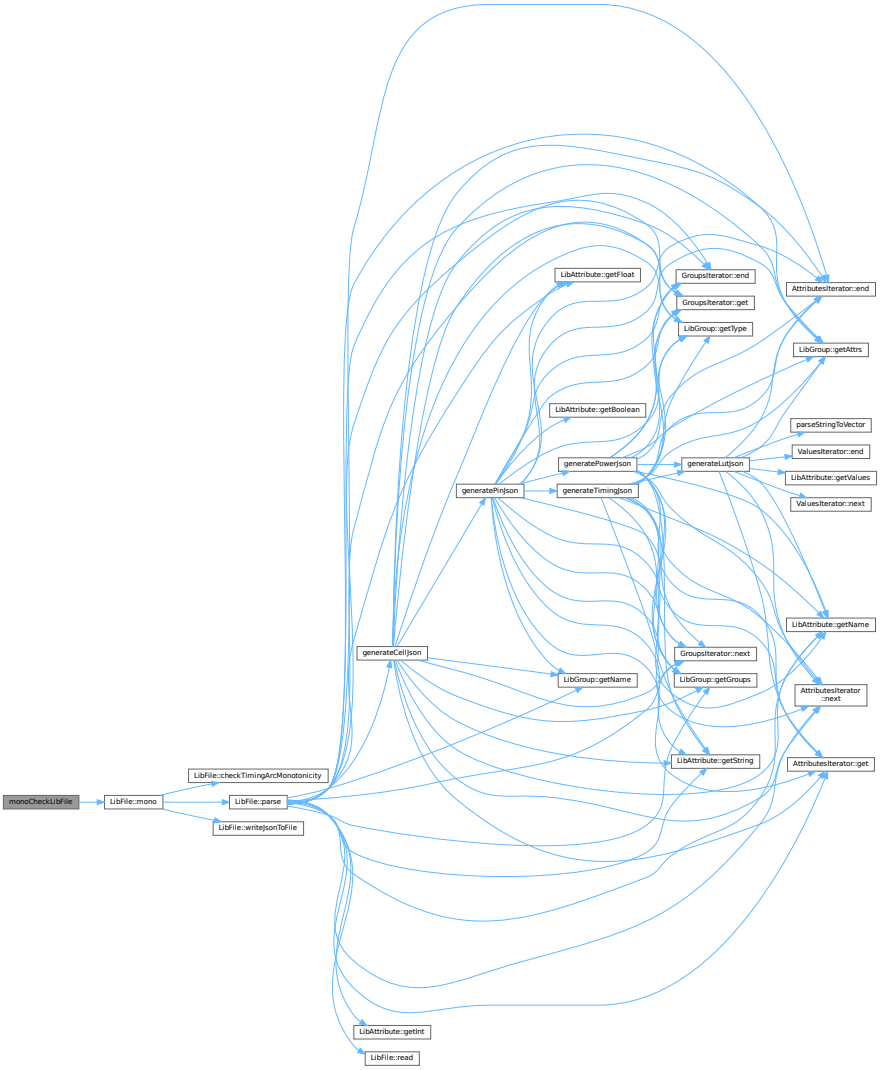
library_path	Path to the library file to check
log_file_name	Name of the log file to create (optional). If empty, a default name will be generated from the library file name
is_slew	Flag indicating whether to check input slew monotonicity (true) or output load monotonicity (false)

Exceptions

The	function catches and logs any exceptions but does not rethrow them

Definition at line 81 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

7.33.1.4 parseLibFile()

```
void parseLibFile (
                const std::string & library_path,
                const std::string log_file_name )
```

Parses a library file and generates corresponding output.

This function processes the given library file, parsing its contents and generating a JSON output. It also logs the parsing process to a specified log file or creates a default log file if none is provided.

Parameters

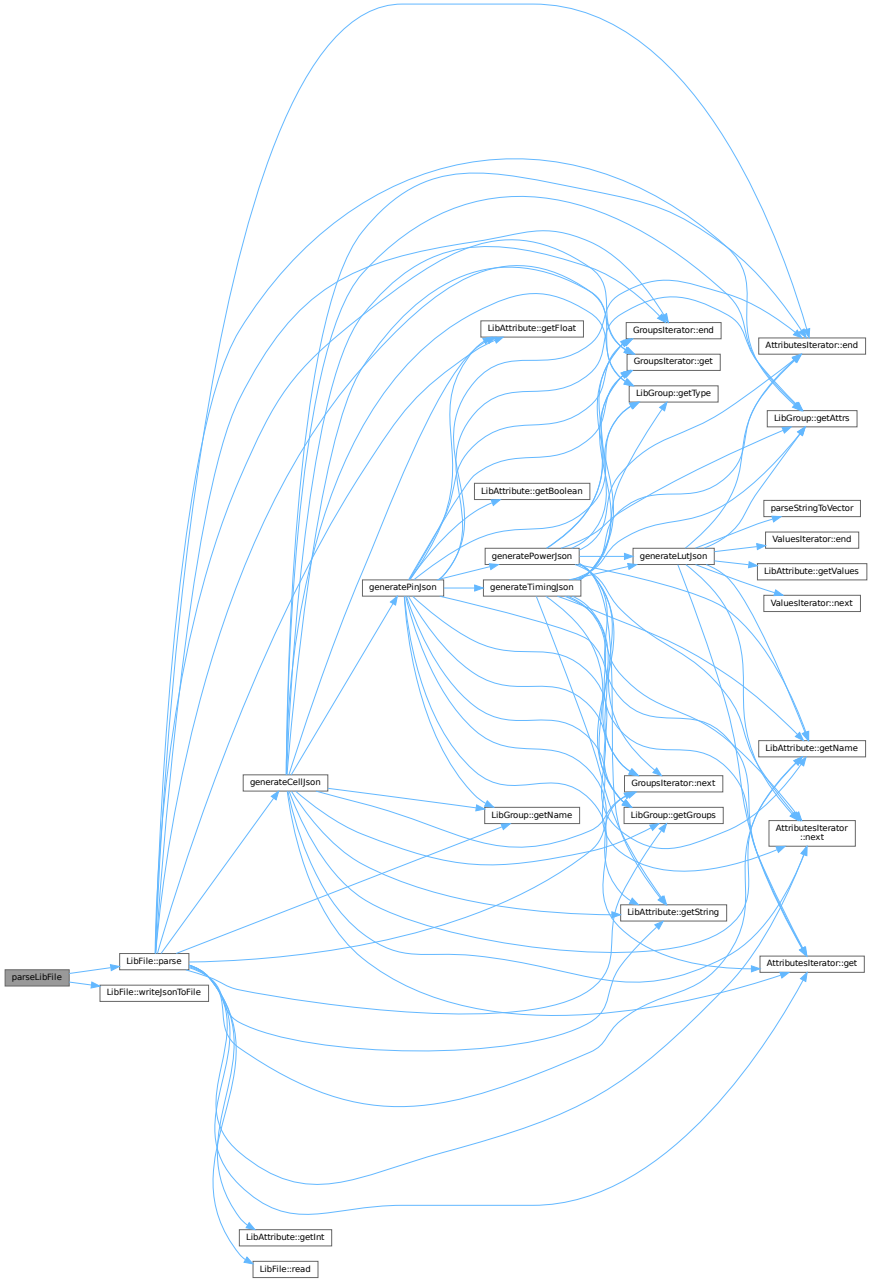
library_path	Path to the library file that needs to be parsed
log_file_name	Optional name for the log file. If empty, a default name is generated based on the library filename with ".parse.log" extension

Exceptions

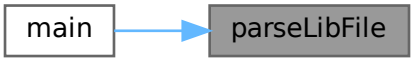
The	function catches and logs any exceptions but doesn't propagate them

Definition at line 49 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.33.1.5 printInfo()

```
void printInfo ( )
```

Sets up and configures the global logger and prints application information.

This function performs the following operations:

1. Creates a console sink for logging with level set to INFO
2. Creates a file sink for logging with level set to TRACE, saving to [APP_NAME].log
3. Configures a logger with both sinks and sets it as the default logger
4. Outputs basic application information:
   - Version and build timestamp
   - Author information
   - Log file location

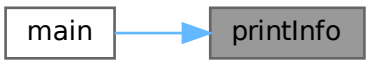
Note

Uses spdlog library for logging functionality

Depends on APP_NAME, APP_VERSION, BUILD_TIMESTAMP, APP_AUTHOR, and APP↩ _CONTACT macros

Definition at line 18 of file LibFileOperations.cpp.

Here is the caller graph for this function:



7.33.1.6 spiceLibFile()

```
void spiceLibFile (
                const std::string & library_path,
                const std::string & log_file_name,
                int chain_length,
                const std::vector< std::string > & cell_names,
                const std::string & verilog_lib_file,
                const std::string & spice_lib_file )
```

Generates a SPICE library file from a given library file, applying a specified chain length and cell names.

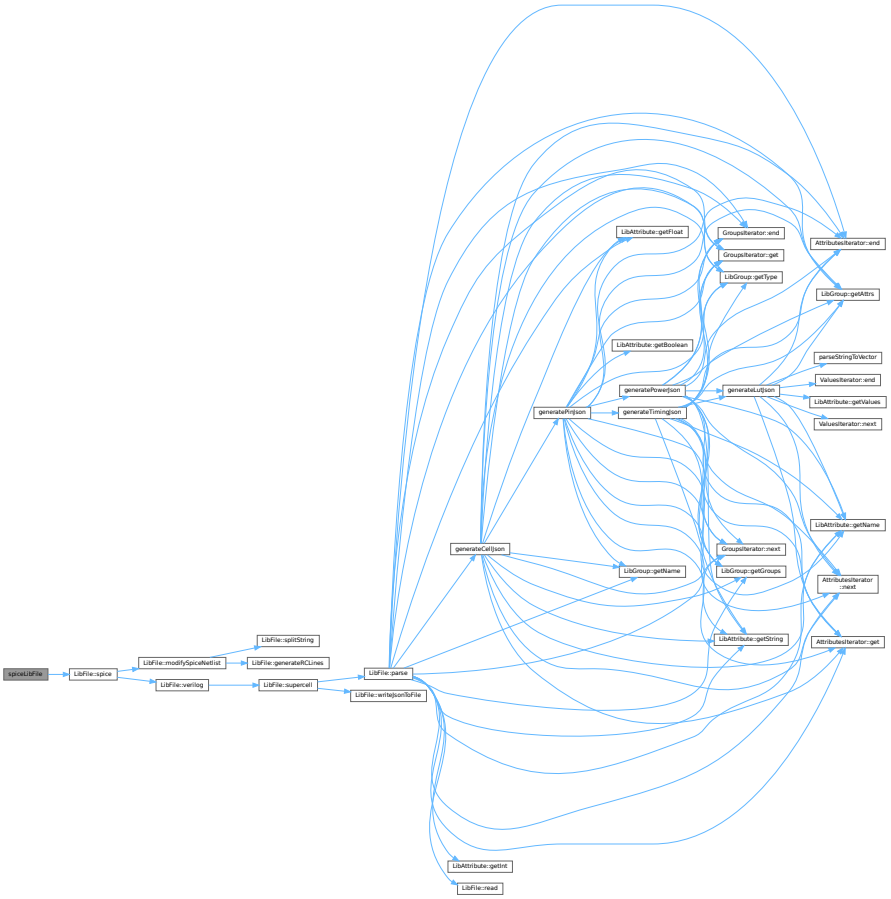
This function takes a library file path, a log file name, a chain length, a vector of cell names, a Verilog library file path, and a SPICE library file path as input. It initializes a LibFile object, validates the chain length, and then calls the spice method of the LibFile object to generate the SPICE library file. It logs the start and end of the SPICE generation process, as well as any errors that occur.

Parameters

library_path	The path to the input library file.
log_file_name	The name of the log file. If empty, a default log file name is generated based on the library file name.
chain_length	The chain length to use during SPICE generation. Must be $>= 1$.
cell_names	A vector of cell names to include in the SPICE generation.
verilog_lib_file	The path to the Verilog library file.
spice_lib_file	The path to the output SPICE library file.

Definition at line 202 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

7.33.1.7 supercellLibFile()

```
void supercellLibFile (
                const std::string & library_path,
                const std::string & log_file_name,
```

```
int chain_length,
const std::vector< std::string > & cell_names )
```

Creates supercell map structures from a Liberty library file.

This function reads a Liberty file, creates supercell structures based on the specified chain length and cell names, and logs the process to a file.

Parameters

library_path	Path to the Liberty file to process
log_file_name	Name of the log file (if empty, defaults to "[library_name].supercell.log")
chain_length	The length of chains to create (must be >= 1)
cell_names	Vector of cell names to process for supercell generation

Exceptions

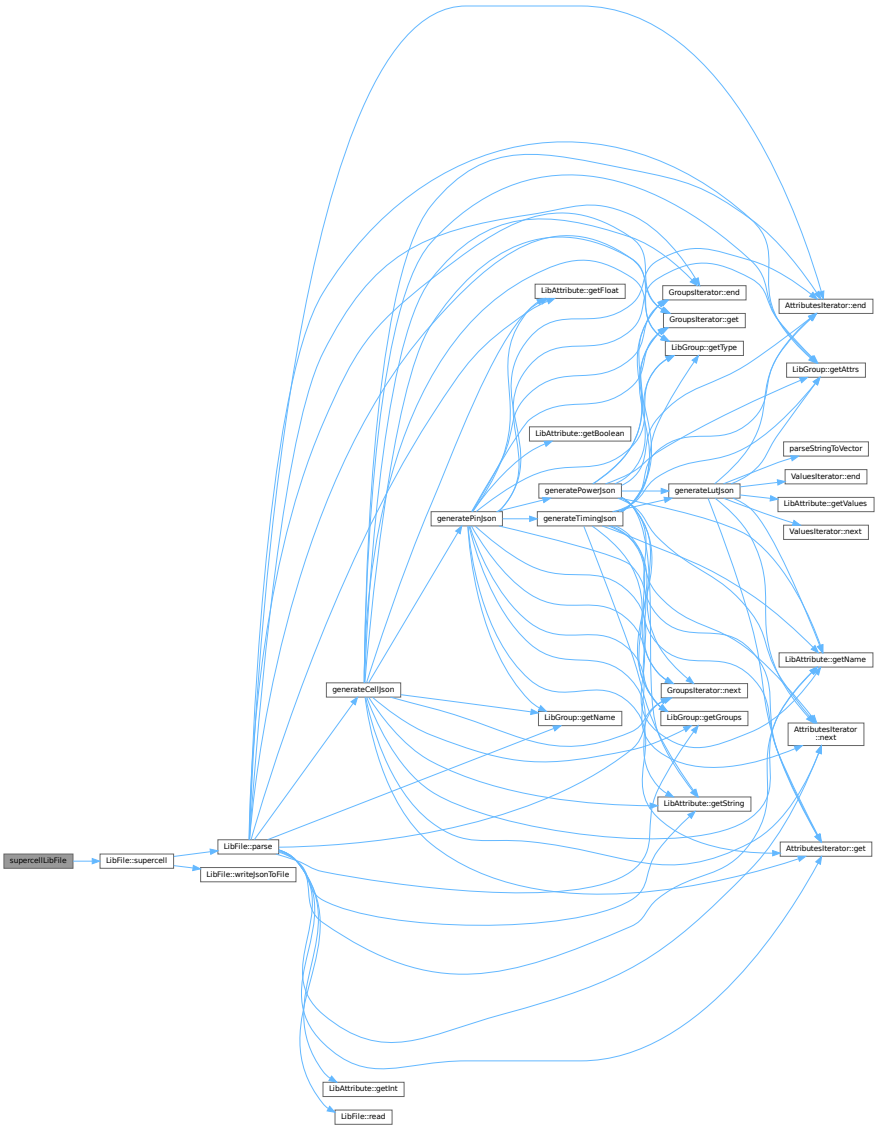
May	pass through exceptions from the LibFile::supercell method

Note

The function validates the chain length and logs all activities including errors that might occur during processing

Definition at line 116 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

7.33.1.8 verilogLibFile()

```
void verilogLibFile (
                const std::string & library_path,
                const std::string & log_file_name,
                int chain_length,
                const std::vector< std::string > & cell_names )
```

Generates Verilog files from a library file for specified cells.

This function processes a library file and generates Verilog representation for the specified cell names with a given chain length. The operation results are logged to a specified or default log file.

Parameters

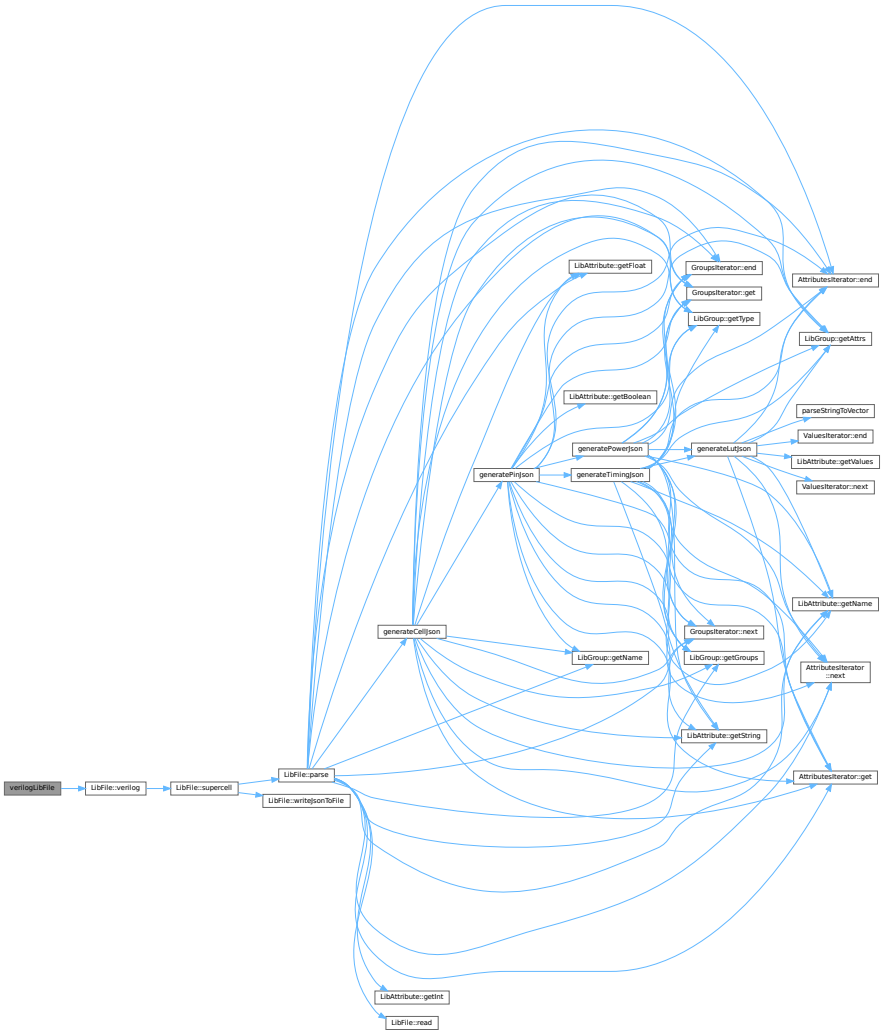
library_path	Path to the library file to process
log_file_name	Name for the log file (if empty, a default name will be generated)
chain_length	Number of cells to chain together, must be >= 1
cell_names	Vector of cell names to generate Verilog for

Exceptions

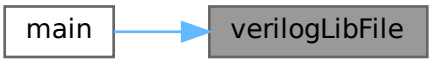
Catches	any exceptions from the verilog generation process and logs them

Definition at line 157 of file LibFileOperations.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34 LibFileOperations.cpp

Go to the documentation of this file.

```
00001 #include "LibFileOperations.hpp"
00002
00018 void printInfo() {
00019   // Set Global Logger
00020   auto console_sink = std::make_shared<spdlog::sinks::stdout_color_sink_mt>();
00021   console_sink->set_level(spdlog::level::info);
00022   std::string app_name = APP_NAME;
00023   auto file_sink = std::make_shared<spdlog::sinks::basic_file_sink_mt>(app_name + ".log", true);
00024   file_sink->set_level(spdlog::level::trace);
00025
00026   std::vector<spdlog::sink_ptr> sinks{console_sink, file_sink};
00027   auto logger = std::make_shared<spdlog::logger>(APP_NAME, sinks.begin(), sinks.end());
00028   logger->set_level(spdlog::level::trace);
00029   spdlog::set_default_logger(logger);
00030
00031   spdlog::info("Version {}, Built: {}", APP_VERSION, BUILD_TIMESTAMP);
00032   spdlog::info("Author: {}, Email: {}", APP_AUTHOR, APP_CONTACT);
00033   spdlog::info("Global log file in: '{}'", app_name + ".log");
00034 }
00035
00049 void parseLibFile(const std::string &library_path, const std::string log_file_name) {
00050   std::string logname = log_file_name.empty()
00051                             ? std::filesystem::path(library_path).stem().string() + ".parse.log"
00052                             : log_file_name;
00053   LibFile libfile(library_path, logname);
00054
00055   libfile.logger_->info("Starting parse for file: '{}' ...", library_path);
00056
00057   try {
00058     libfile.parse();
00059     libfile.writeJsonToFile();
00060     libfile.logger_->info("Successfully parsed file: '{}'", library_path);
00061   } catch (const std::exception &e) {
00062     libfile.logger_->error("Error parsing file '{}': {}", library_path, e.what());
00063   }
00064 }
00065
00081 void monoCheckLibFile(const std::string &library_path, const std::string log_file_name,
00082                       bool is_slew) {
00083   std::string logname = log_file_name.empty()
00084                             ? std::filesystem::path(library_path).stem().string() + ".mono.log"
00085                             : log_file_name;
00086   LibFile libfile(library_path, logname);
00087
00088   libfile.logger_->info("Starting monotonicity check for file: '{}', input slew: {}", library_path,
00089                         is_slew);
00090
00091   try {
00092     libfile.mono(is_slew);
00093     libfile.logger_->info("Successfully completed monotonicity check for file: '{}'", library_path);
00094   } catch (const std::exception &e) {
00095     libfile.logger_->error("Error during monotonicity check for file '{}': {}", library_path,
00096                            e.what());
00097   }
00098 }
00099
00116 void supercellLibFile(const std::string &library_path, const std::string &log_file_name,
00117                       int chain_length, const std::vector<std::string> &cell_names) {
00118   std::string logname = log_file_name.empty()
00119                             ? std::filesystem::path(library_path).stem().string() + ".supercell.log"
00120                             : log_file_name;
00121   LibFile libfile(library_path, logname);
00122
00123   // Check chain length validity
00124   if (chain_length < 1) {
00125     libfile.logger_->error("Invalid chain length: {}. Chain length must be >= 1.", chain_length);
00126     return;
00127   }
00128   std::stringstream ss;
```

```
00129   for (const auto &cell : cell_names) {
00130     ss << cell << ", ";
00131   }
00132   libfile.logger_->info("Starting supercell generation for file: '{}', chain length: {}, cells: {}",
00133                         library_path, chain_length, ss.str());
00134   try {
00135     libfile.supercell(chain_length, cell_names);
00136     libfile.logger_->info("Successfully generated supercells for file: '{}'", library_path);
00137   } catch (const std::exception &e) {
00138     libfile.logger_->error("Error during supercell generation for file '{}': {}", library_path,
00139                            e.what());
00140   }
00141 }
00142
00157 void verilogLibFile(const std::string &library_path, const std::string &log_file_name,
00158                     int chain_length, const std::vector<std::string> &cell_names) {
00159   std::string logname = log_file_name.empty()
00160                             ? std::filesystem::path(library_path).stem().string() + ".verilog.log"
00161                             : log_file_name;
00162   LibFile libfile(library_path, logname);
00163
00164   // Check chain length validity
00165   if (chain_length < 1) {
00166     libfile.logger_->error("Invalid chain length: {}. Chain length must be >= 1.", chain_length);
00167     return;
00168   }
00169   std::stringstream ss;
00170   for (const auto &cell : cell_names) {
00171     ss << cell << ", ";
00172   }
00173   libfile.logger_->info("Starting Verilog generation for file: '{}', chain length: {}, cells: {}",
00174                         library_path, chain_length, ss.str());
00175   try {
00176     libfile.verilog(chain_length, cell_names);
00177     libfile.logger_->info("Successfully generated Verilog for file: '{}'", library_path);
00178   } catch (const std::exception &e) {
00179     libfile.logger_->error("Error during Verilog generation for file '{}': {}", library_path,
00180                            e.what());
00181   }
00182 }
00183
00202 void spiceLibFile(const std::string &library_path, const std::string &log_file_name,
00203                   int chain_length, const std::vector<std::string> &cell_names,
00204                   const std::string &verilog_lib_file, const std::string &spice_lib_file) {
00205   std::string logname = log_file_name.empty()
00206                             ? std::filesystem::path(library_path).stem().string() + ".spice.log"
00207                             : log_file_name;
00208   LibFile libfile(library_path, logname);
00209
00210   // Check chain length validity
00211   if (chain_length < 1) {
00212     libfile.logger_->error("Invalid chain length: {}. Chain length must be >= 1.", chain_length);
00213     return;
00214   }
00215   std::stringstream ss;
00216   for (const auto &cell : cell_names) {
00217     ss << cell << ", ";
00218   }
00219   libfile.logger_->info("Starting SPICE generation for file: '{}', chain length: {}, cells: {}",
00220                         library_path, chain_length, ss.str());
00221   try {
00222     libfile.spice(chain_length, cell_names, verilog_lib_file, spice_lib_file);
00223     libfile.logger_->info("Successfully generated SPICE for file: '{}'", library_path);
00224   } catch (const std::exception &e) {
00225     libfile.logger_->error("Error during SPICE generation for file '{}': {}", library_path,
00226                            e.what());
00227   }
00228 }
00229
```

```
00249 void compareLibFiles(const std::string &ref_lib, const std::string &comp_lib, const double reltol,
00250                       const double abstol, std::string &report_file_name) {
00251   std::string ref_logname = std::filesystem::path(ref_lib).stem().string() + ".cmp.log";
00252   std::string comp_logname = std::filesystem::path(comp_lib).stem().string() + ".cmp.log";
00253   LibFile ref_libfile(ref_lib, ref_logname), comp_libfile(comp_lib, comp_logname);
00254
00255   // Check relative tolerance validity
00256   if (reltol < 0.0) {
00257     spdlog::error("Invalid relative tolerance: {}. Relative tolerance must be >= 0.0.", reltol);
00258     return;
00259   }
00260
00261   // Check the report file name
00262   if (report_file_name.empty()) {
00263     report_file_name = comp_libfile.basename_ + ".cmp.md";
00264   } else if (std::filesystem::path(report_file_name).extension() != ".txt" &&
00265              std::filesystem::path(report_file_name).extension() != ".md") {
00266     // report file name not end in .txt or .md, warn user and append .md
00267     report_file_name = report_file_name + ".md";
00268     spdlog::warn("Report file name does not end in .txt or .md. Report will be written to '{}'",
00269                  report_file_name);
00270   }
00271
00272   LibraryComparator comparator(ref_libfile, comp_libfile, reltol, abstol);
00273   comparator.generateReport(report_file_name);
00274   spdlog::info("Comparison completed. Report written to: '{}'", report_file_name);
00275 }
00276
00308 void funcLibFile(const std::string &ref_file, const std::string &comp_file,
00309                  const std::vector<std::string> &cell_names, std::string &report_file_name) {
00310   // Check if the files are Liberty or Verilog
00311   std::string ref_ext = std::filesystem::path(ref_file).extension();
00312   std::string comp_ext = std::filesystem::path(comp_file).extension();
00313   std::string ref_logname = std::filesystem::path(ref_file).stem().string() + ".cmp.log";
00314   std::string comp_logname = std::filesystem::path(comp_file).stem().string() + ".cmp.log";
00315
00316   // Pre cell names check
00317   if (cell_names.empty()) {
00318     spdlog::error("No cell names provided for functional equivalence check.");
00319     return;
00320   }
00321
00322   // Check the report file name
00323   if (report_file_name.empty()) {
00324     report_file_name = std::filesystem::path(comp_file).stem().string() + ".logic.cmp.md";
00325   } else if (std::filesystem::path(report_file_name).extension() != ".txt" &&
00326              std::filesystem::path(report_file_name).extension() != ".md") {
00327     // report file name not end in .txt or .md, warn user and append .md
00328     report_file_name = report_file_name + ".md";
00329     spdlog::warn("Report file name does not end in .txt or .md. Report will be written to '{}'",
00330                  report_file_name);
00331   }
00332
00333   // Clear the report file
00334   std::ofstream report_file(report_file_name);
00335   if (!report_file.is_open()) {
00336     spdlog::error("Failed to open report file: '{}'", report_file_name);
00337     return;
00338   }
00339   report_file.close();
00340   spdlog::info("Report file cleared: '{}'", report_file_name);
00341
00342   // Declare LibFile objects as pointers, initialized to nullptr
00343   LibFile *ref_libfile = nullptr;
00344   LibFile *comp_libfile = nullptr;
00345
00346   // Check the reference file extension
00347   if (ref_ext == ".v") {
00348     spdlog::info("Reference file in Verilog format.");
```

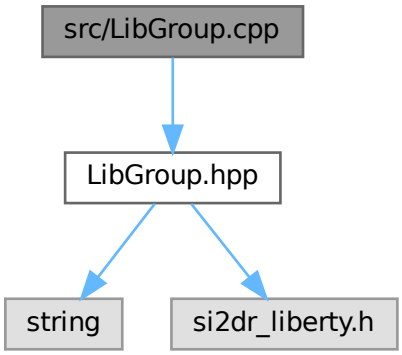
```
00349   } else if (ref_ext == ".lib") {
00350     spdlog::info("Reference file in Liberty format.");
00351     ref_libfile = new LibFile(ref_file, ref_logname);
00352   } else {
00353     spdlog::error("Unsupported reference file format: '{}'", ref_ext);
00354     return;
00355   }
00356   // Check the comparison file extension
00357   if (comp_ext == ".v") {
00358     spdlog::info("Comparison file in Verilog format.");
00359   } else if (comp_ext == ".lib") {
00360     spdlog::info("Comparison file in Liberty format.");
00361     comp_libfile = new LibFile(comp_file, comp_logname);
00362   } else {
00363     spdlog::error("Unsupported comparison file format: '{}'", comp_ext);
00364     return;
00365   }
00366
00367   // Tranverse the cell names and check functional equivalence
00368   for (const auto &cell : cell_names) {
00369     spdlog::info("Checking functional equivalence for cell: '{}'", cell);
00370     std::map<std::string, std::string> ref_outpin_map;
00371     std::map<std::string, std::string> comp_outpin_map;
00372
00373     if (ref_ext == ".v") {
00374       ref_outpin_map = extractLogicFromVerilog(ref_file, cell);
00375     } else if (ref_ext == ".lib") {
00376       ref_outpin_map = ref_libfile->logic(cell);
00377     }
00378
00379     if (comp_ext == ".v") {
00380       // getAST(comp_file, cell);
00381       // extractAndPrintNetlistInfo(comp_file, cell);
00382       comp_outpin_map = extractLogicFromVerilog(comp_file, cell);
00383     } else if (comp_ext == ".lib") {
00384       comp_outpin_map = comp_libfile->logic(cell);
00385     }
00386
00387     spdlog::info("Logic function expressions collected for cell: '{}'", cell);
00388     spdlog::info("Starting logic comparison ...");
00389
00390     LogicComparator comparator(ref_outpin_map, comp_outpin_map, cell);
00391     // Easter egg
00392     if (cell == "easteregg") {
00393       spdlog::info("Easter egg  found!   Running ExprTk Eample 07...");
00394       comparator.logic();
00395       return;
00396     }
00397
00398     // // Test for preprocessing
00399     // std::string ref_expr;
00400     // std::string comp_expr;
00401     // for (const auto &pin : ref_outpin_map) {
00402     //   spdlog::info("Pin -> Expression: {} -> {}", pin.first, pin.second);
00403     //   ref_expr = comparator.preprocessExpression(pin.second);
00404     // }
00405     // for (const auto &pin : comp_outpin_map) {
00406     //   spdlog::info("Pin => Expression: {} => {}", pin.first, pin.second);
00407     //   comp_expr = comparator.preprocessExpression(pin.second);
00408     // }
00409     // std::vector<std::string> sorted_vars;
00410     // comparator.extractVariables(ref_expr, comp_expr, sorted_vars);
00411     // struct PinComparisonResult pin_comparison_result;
00412     // // comp_expr = "not ((not(A1) and not(A2)) or B)";
00413     // comparator.compareSingleExpressionPair(ref_expr, comp_expr, sorted_vars,
00414     //                                        pin_comparison_result);
00415
00416     // comparator.compareCellLogic();
00417     comparator.compareCellLogic();
```

```
00418     comparator.generateReport(report_file_name);
00419
00420     spdlog::info("Functional equivalence check completed for cell: '{}'", cell);
00421   }
00422   spdlog::info("Functional equivalence check completed for all cells,");
00423   spdlog::info("Report written to: '{}'", report_file_name);
00424 }
```

7.35 src/LibGroup.cpp File Reference

`#include "LibGroup.hpp"`

Include dependency graph for LibGroup.cpp:



7.36 LibGroup.cpp

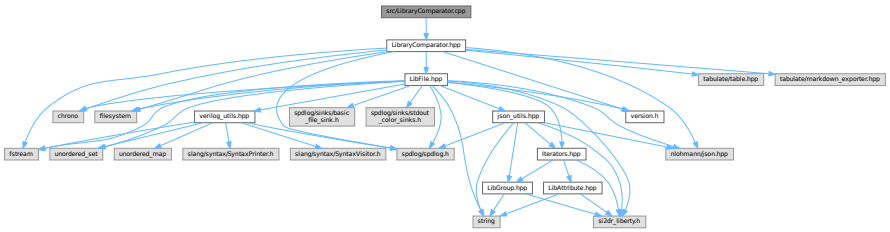
Go to the documentation of this file.

```
00001 #include "LibGroup.hpp"
00002
00003 LibGroup::LibGroup(si2drGroupIdT group, si2drErrorT &err) : group_(group), err_(err) {}
00004
00005 LibGroup::~LibGroup() {}
00006
00007 std::string LibGroup::getName() {
00008   si2drNamesIdT names = si2drGroupGetNames(group_, &err_);
00009   si2drStringT name = si2drIterNextName(names, &err_);
00010   si2drIterQuit(names, &err_);
00011   return name ? std::string(name) : std::string();
00012 }
00013
00014 std::string LibGroup::getType() {
00015   si2drStringT type = si2drGroupGetGroupType(group_, &err_);
00016   return type ? std::string(type) : std::string();
00017 }
00018
00019 si2drAttrsIdT LibGroup::getAttrs() { return si2drGroupGetAttrs(group_, &err_); }
00020
00021 si2drGroupsIdT LibGroup::getGroups() { return si2drGroupGetGroups(group_, &err_); }
```

7.37 src/LibraryComparator.cpp File Reference

`#include "LibraryComparator.hpp"`

Include dependency graph for LibraryComparator.cpp:



7.38 LibraryComparator.cpp

Go to the documentation of this file.

```
00001 #include "LibraryComparator.hpp"
00002
00021 LibraryComparator::LibraryComparator(LibFile &ref_libfile, LibFile &comp_libfile, double reltol,
00022                                       double abstol)
00023     : reltol_(reltol), abstol_(abstol) {
00024   std::string ref_json_name = ref_libfile.basename_ + ".json";
00025   if (!std::filesystem::exists(ref_json_name)) {
00026     spdlog::info("Reference JSON file {} not found. Parsing first.", ref_json_name);
00027     ref_libfile.parse();
00028     ref_libfile.writeJsonToFile();
00029     ref_json_ = ref_libfile.lib_json_;
00030   } else {
00031     // Read the JSON file into json object
00032     std::ifstream ref_in(ref_json_name);
00033     if (!ref_in.is_open()) {
00034       spdlog::error("Could not open file '{}' for reading", ref_json_name);
00035       return;
00036     }
00037     try {
00038       ref_json_ = json::parse(ref_in);
00039     } catch (const json::parse_error &e) {
00040       spdlog::error("Error parsing reference JSON file '{}': {}", ref_json_name, e.what());
00041       return;
00042     }
00043   }
00044   std::string comp_json_name = comp_libfile.basename_ + ".json";
00045   if (!std::filesystem::exists(comp_json_name)) {
00046     spdlog::info("Comparison JSON file {} not found. Parsing first.", comp_json_name);
00047     comp_libfile.parse();
00048     comp_libfile.writeJsonToFile();
00049     comp_json_ = comp_libfile.lib_json_;
00050   } else {
00051     // Read the JSON file into json object
00052     std::ifstream comp_in(comp_json_name);
00053     if (!comp_in.is_open()) {
00054       spdlog::error("Could not open file '{}' for reading", comp_json_name);
00055       return;
00056     }
00057     try {
00058       comp_json_ = json::parse(comp_in);
00059     } catch (const json::parse_error &e) {
```

```
00060       spdlog::error("Error parsing comparison JSON file '{}': {}", comp_json_name, e.what());
00061       return;
00062     }
00063   }
00064
00065   ref_lib_path_ = ref_libfile.filepath_;
00066   comp_lib_path_ = comp_libfile.filepath_;
00067   spdlog::info("Successfully loaded JSON files for comparison");
00068 }
00069
00098 void LibraryComparator::compareLut(const std::string &cell_name, const std::string &pin_name,
00099                                    const std::string &timing_type, const std::string &related_pin,
00100                                    const std::string &arc_name, const json &ref_lut,
00101                                    const json &comp_lut, Table &table) {
00102   spdlog::info("Comparing LUT of '{}' ...", arc_name);
00103   std::vector<double> ref_index_1 = ref_lut["index_1"].get<std::vector<double>>();
00104   std::vector<double> ref_index_2 = ref_lut["index_2"].get<std::vector<double>>();
00105   std::vector<double> comp_index_1 = comp_lut["index_1"].get<std::vector<double>>();
00106   std::vector<double> comp_index_2 = comp_lut["index_2"].get<std::vector<double>>();
00107   if (ref_index_1 != comp_index_1 || ref_index_2 != comp_index_2) {
00108     spdlog::warn("Mismatch in LUT indices for '{}' in cell: '{}', pin: '{}', related_pin: '{}', "
00109                  "timing_type: '{}'",
00110                  arc_name, cell_name, pin_name, related_pin, timing_type);
00111     return;
00112   } else {
00113     spdlog::debug("LUT indices match for '{}'", arc_name);
00114
00115     std::vector<std::vector<double>> comp_value_matrix;
00116     // Generate a matrix of values from the JSON array for comparison library
00117     for (const auto &row_val : comp_lut["values"]) {
00118       std::vector<double> row_data;
00119       // Check if the row value is an array
00120       if (!row_val.is_array()) {
00121         spdlog::error(
00122             "Invalid LUT format: '{}' values should be an array of arrays in comp_lib '{}', "
00123             "cell '{}', pin '{}', "
00124             "related_pin '{}', timing_type '{}'",
00125             arc_name, this->comp_lib_path_.string(), cell_name, pin_name, related_pin, timing_type);
00126         return;
00127       }
00128       // Check if non-numeric values
00129       for (const auto &val : row_val) {
00130         if (val.is_number()) {
00131           row_data.push_back(val.get<double>());
00132         } else {
00133           spdlog::error("Non-numeric value found in '{}'.values, skipping value: {} in comp_lib "
00134                         "'{}', cell '{}', "
00135                         "pin '{}', related_pin '{}', timing_type '{}'",
00136                         arc_name, val.dump(), this->comp_lib_path_.string(), cell_name, pin_name,
00137                         related_pin, timing_type);
00138           return;
00139         }
00140       }
00141       comp_value_matrix.push_back(row_data);
00142     }
00143     // Generate a matrix of values from the JSON array for reference library
00144     std::vector<std::vector<double>> ref_value_matrix;
00145     for (const auto &row_val : ref_lut["values"]) {
00146       std::vector<double> row_data;
00147       // Check if the row value is an array
00148       if (!row_val.is_array()) {
00149         spdlog::error(
00150             "Invalid LUT format: '{}' values should be an array of arrays in ref_lib '{}', "
00151             "cell '{}', pin '{}', "
00152             "related_pin '{}', timing_type '{}'",
00153             arc_name, this->ref_lib_path_.string(), cell_name, pin_name, related_pin, timing_type);
00154         return;
00155       }
00156       // Check if non-numeric values
```

```
00157       for (const auto &val : row_val) {
00158         if (val.is_number()) {
00159           row_data.push_back(val.get<double>());
00160         } else {
00161           spdlog::error("Non-numeric value found in '{}'.values, skipping value: {} in ref_lib "
00162                         "'{}', cell '{}', "
00163                         "pin '{}', related_pin '{}', timing_type '{}'",
00164                         arc_name, val.dump(), this->ref_lib_path_.string(), cell_name, pin_name,
00165                         related_pin, timing_type);
00166           return;
00167         }
00168       }
00169       ref_value_matrix.push_back(row_data);
00170     }
00171     // Compare the matrix for relative tolerance
00172     if (!comp_value_matrix.empty() && !comp_value_matrix[0].empty()) {
00173       for (size_t i = 0; i < comp_value_matrix.size(); ++i) {
00174         for (size_t j = 0; j < comp_value_matrix[i].size(); ++j) {
00175           double ref_val = ref_value_matrix[i][j];
00176           double comp_val = comp_value_matrix[i][j];
00177           if (std::abs(ref_val - comp_val) > reltol_ * std::abs(ref_val) &&
00178               std::abs(ref_val - comp_val) > abstol_) {
00179             spdlog::debug("LUT value mismatch for '{}', index_1: {}, index_2: {}", arc_name,
00180                           ref_index_1[i], ref_index_2[j]);
00181             table.add_row({related_pin + "->" + pin_name, std::to_string(ref_val),
00182                            std::to_string(comp_val), std::to_string(comp_val - ref_val),
00183                            std::to_string((comp_val - ref_val) / ref_val * 100), timing_type,
00184                            arc_name, std::to_string(i + 1), std::to_string(ref_index_1[i]),
00185                            std::to_string(j + 1), std::to_string(ref_index_2[j]), "<"});
00186             spdlog::debug("table size: {}", table.size());
00187           } else {
00188             spdlog::debug("LUT value match for '{}', index_1: {}, index_2: {}", arc_name,
00189                           ref_index_1[i], ref_index_2[j]);
00190           }
00191         }
00192       }
00193     } else {
00194       spdlog::error(
00195           "Empty or invalid 'values' array found for cell: '{}', pin: '{}', related_pin: '{}', "
00196           "timing_type: '{}', arc: '{}'",
00197           cell_name, pin_name, related_pin, timing_type, arc_name);
00198     }
00199   }
00200 }
00201
00220 void LibraryComparator::compareTimingArc(const std::string &cell_name, const std::string &pin_name,
00221                                          const std::string &timing_type, const json &ref_timing_arc,
00222                                          const json &comp_timing_arc, Table &table) {
00223   spdlog::info("Comparing timing type '{}' ...", timing_type);
00224
00225   std::string related_pin = ref_timing_arc["related_pin"].get<std::string>();
00226   spdlog::debug("Related pin: '{}'", related_pin);
00227
00228   std::vector<std::string> arc_names = {"cell_rise", "cell_fall", "rise_transition",
00229                                         "fall_transition"};
00230   for (auto arc_name : arc_names) {
00231     if (comp_timing_arc.contains(arc_name)) {
00232       spdlog::debug("Comparing timing arc: '{}'", arc_name);
00233       // Look for the arc in the reference JSON
00234       if (ref_timing_arc.contains(arc_name)) {
00235         // Found this arc
00236         spdlog::debug("Found timing arc: '{}'", arc_name);
00237         compareLut(cell_name, pin_name, timing_type, related_pin, arc_name,
00238                    ref_timing_arc[arc_name], comp_timing_arc[arc_name], table);
00239       } else {
00240         // arc not found in reference JSON
00241         spdlog::warn("Timing arc: '{}' not found in reference JSON", arc_name);
00242       }
00243     }
```

```
00244   }
00245 }
00246
00261 void LibraryComparator::comparePin(const std::string &cell_name, const std::string &pin_name,
00262                                    const json &ref_pin, const json &comp_pin, Table &table) {
00263   spdlog::info("Comparing pin '{}' ...", pin_name);
00264
00265   if (comp_pin.contains("timing_arcs")) {
00266     for (const auto &comp_arc : comp_pin["timing_arcs"]) {
00267       std::string timing_type = comp_arc["timing_type"].get<std::string>();
00268       spdlog::debug("Comparing timing type: '{}'", timing_type);
00269       // Look for the timing type in the reference JSON
00270       auto ref_arc_it = std::find_if(
00271           ref_pin["timing_arcs"].begin(), ref_pin["timing_arcs"].end(),
00272           [&timing_type](const json &ref_arc) { return ref_arc["timing_type"] == timing_type; });
00273       if (ref_arc_it != ref_pin["timing_arcs"].end()) {
00274         // Found this timing type
00275         spdlog::debug("Found timing type: '{}'", timing_type);
00276         compareTimingArc(cell_name, pin_name, timing_type, *ref_arc_it, comp_arc, table);
00277       } else {
00278         // timing arc not found in reference JSON
00279         spdlog::warn("Timing arc: '{}' not found in reference JSON", timing_type);
00280       }
00281     }
00282   } else {
00283     spdlog::info("No timing arcs found for pin: '{}'", comp_pin["pin_name"].get<std::string>());
00284   }
00285 }
00286
00301 void LibraryComparator::compareCell(const std::string &cell_name, const json &ref_cell,
00302                                     const json &comp_cell, Table &table) {
00303   spdlog::info("Comparing cell '{}' ...", cell_name);
00304
00305   if (comp_cell.contains("output_pins")) {
00306     for (const auto &comp_pin : comp_cell["output_pins"]) {
00307       std::string pin_name = comp_pin["pin_name"].get<std::string>();
00308       spdlog::debug("Comparing pin: '{}'", pin_name);
00309       // Look for the pin in the reference JSON
00310       auto ref_pin_it = std::find_if(
00311           ref_cell["output_pins"].begin(), ref_cell["output_pins"].end(),
00312           [&pin_name](const json &ref_pin) { return ref_pin["pin_name"] == pin_name; });
00313       if (ref_pin_it != ref_cell["output_pins"].end()) {
00314         // Found this pin
00315         spdlog::debug("Found pin: '{}'", pin_name);
00316         comparePin(cell_name, pin_name, *ref_pin_it, comp_pin, table);
00317       } else {
00318         // pin not found in reference JSON
00319         spdlog::warn("Pin: '{}' not found in reference JSON", pin_name);
00320       }
00321     }
00322   } else {
00323     spdlog::info("No output pins found for cell: '{}'", cell_name);
00324   }
00325 }
00326
00343 void LibraryComparator::generateReport(const std::string &output_file) {
00344   std::ofstream outfile(output_file);
00345   outfile << "# LIBRARY comparison\n" << std::endl;
00346   outfile << "**Reference library: " << ref_lib_path_ << "**\n" << std::endl;
00347   outfile << "**Comparison library: " << comp_lib_path_ << "**\n" << std::endl;
00348   outfile << "**Absolute tolerance: " << abstol_ << "**\n" << std::endl;
00349   outfile << "**Relative tolerance: " << reltol_ << "**\n" << std::endl;
00350   outfile << "**Performed by " << APP_NAME << " v" << APP_VERSION << " from " << APP_AUTHOR;
00351   auto now = std::chrono::system_clock::now();
00352   std::time_t now_time_t = std::chrono::system_clock::to_time_t(now);
00353   std::tm *now_tm = std::localtime(&now_time_t);
00354   outfile << ". on: " << std::put_time(now_tm, "%c") << "**\n" << std::endl;
00355   outfile << "> Legend: < outlier, * scaled, ! indices switched, ^ slews extrapolated, ~ loads "
00356              "extrapolated,\n";
```

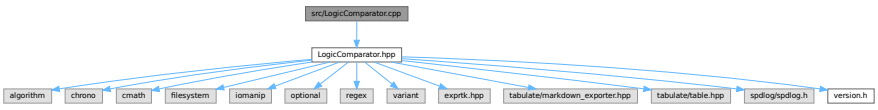
```
00357   outfile << "> \n";
00358   outfile << "> Legend: + padding added, /0 divide by zero, / slews interpolated, # loads "
00359              "interpolated\n";
00360   outfile << "> \n";
00361   outfile << "> Legend: << value is less but unknown, >> value is more but unknown\n\n";
00362
00363   spdlog::info("Starting comparison report generation ...");
00364   for (const auto &comp_cell : comp_json_["cells"]) {
00365     std::string cell_name = comp_cell["cell_name"].get<std::string>();
00366     spdlog::debug("Comparing cell: '{}'", cell_name);
00367     // Look for the cell in the reference JSON
00368     auto ref_cell_it = std::find_if(
00369         ref_json_["cells"].begin(), ref_json_["cells"].end(),
00370         [&cell_name](const json &ref_cell) { return ref_cell["cell_name"] == cell_name; });
00371     if (ref_cell_it != ref_json_["cells"].end()) {
00372       // Found this cell
00373       spdlog::debug("Found cell: '{}'", cell_name);
00374       Table table;
00375       table.add_row({"Pin Name", "Reference", "Comparison", "Diff", "Diff %", "Type", "Arc Name",
00376                      "Row #", "Index_1", "Col #", "Index_2", "Note"});
00377       // Header formatting
00378       for (size_t i = 0; i < table[0].size(); ++i) {
00379         table[0][i].format().font_color(Color::yellow).font_style({FontStyle::bold});
00380       }
00381       compareCell(cell_name, *ref_cell_it, comp_cell, table);
00382
00383       if (table.size() > 1) {
00384         // Data starts from row 1, row 0 is header
00385         size_t failed_count = table.size() - 1;
00386         double sum_diff = 0.0;
00387         double sum_diff_percent = 0.0;
00388         size_t outlier_count = 0;
00389         double max_diff = 0;
00390         double max_diff_percent = 0;
00391         std::vector<double> diff_values; // Store diff values for std deviation calculation
00392
00393         // Index of columns
00394         const int diff_index = 3;
00395         const int diff_percent_index = 4;
00396         const int note_index = 11;
00397
00398         for (size_t i = 1; i < table.size(); ++i) {
00399           try {
00400             double diff = std::stod(table[i][diff_index].get_text());
00401             double diff_percent = std::stod(table[i][diff_percent_index].get_text());
00402
00403             sum_diff += diff;
00404             sum_diff_percent += diff_percent;
00405             diff_values.push_back(diff_percent);
00406
00407             if (table[i][note_index].get_text() == "<") {
00408               outlier_count++;
00409             }
00410             // Change to absolute value for max diff
00411             if (std::abs(diff) > std::abs(max_diff)) {
00412               max_diff = diff;
00413               max_diff_percent = diff_percent;
00414             }
00415
00416           } catch (const std::invalid_argument &e) {
00417             spdlog::error("Could not convert value to double: {}", e.what());
00418             continue; // Skip this row if there's an error
00419           }
00420         }
00421
00422         double avg_diff = sum_diff / failed_count;
00423         double avg_diff_percent = sum_diff_percent / failed_count;
00424
00425         outfile << "## " << cell_name << "\n\n";
```

```
00426         outfile << "Delay Comparison in ns\n\n";
00427         if (output_file.substr(output_file.size() - 3) == ".md") {
00428           MarkdownExporter exporter;
00429           auto markdown = exporter.dump(table);
00430           outfile << markdown << std::endl;
00431         } else {
00432           outfile << table << std::endl;
00433         }
00434         outfile << std::endl;
00435         std::cout << table << std::endl;
00436         outfile << cell_name << " delay SUMMARY (abstol: " << abstol_ << ", reltol: " << reltol_
00437                 << ")\n";
00438 
00439         // Create summary table
00440         Table summary_table;
00441         summary_table.add_row({"Cell Name", "Data Type", "Failed Count", "Avg Diff", "Avg Diff%",
00442                                "Max Diff", "Max Diff%", "Outliers"});
00443         summary_table.add_row({cell_name, "delay(ns)", std::to_string(failed_count),
00444                                std::to_string(avg_diff), std::to_string(avg_diff_percent) + "%",
00445                                std::to_string(max_diff), std::to_string(max_diff_percent) + "%",
00446                                std::to_string(outlier_count)});
00447 
00448         // Output summary table
00449         // Use std::filesystem to reliably get the extension
00450         std::filesystem::path file_path(output_file);
00451         if (file_path.has_extension() && file_path.extension() == ".md") {
00452           MarkdownExporter exporter;
00453           auto markdown = exporter.dump(summary_table);
00454           outfile << markdown << std::endl;
00455         } else {
00456           outfile << summary_table << std::endl;
00457         }
00458         outfile << "Worst delay outlier: Max Abs: " << max_diff << ", Max Rel: " << max_diff_percent
00459                 << "%, Outliers: " << outlier_count << "\n\n";
00460       }
00461     } else {
00462       // cell not found in reference JSON
00463       spdlog::warn("Cell: '{}' not found in reference JSON", cell_name);
00464     }
00465   }
00466 
00467   outfile.close();
00468 }
```

7.39 src/LogicComparator.cpp File Reference

`#include "LogicComparator.hpp"`

Include dependency graph for LogicComparator.cpp:



Functions

- bool isIdentifier (const std::string &token)

Checks if a given string is a valid identifier.

- bool isOperator (const std::string &token)

  Checks if a given token is a logical operator.

7.39.1 Function Documentation

7.39.1.1 isIdentifier()

```
bool isIdentifier (
                const std::string & token )
```

Checks if a given string is a valid identifier.

A valid identifier must:

- Not be empty.
- Consist of alphanumeric characters and underscores.
- Start with an uppercase alphabetic character.
- Contain at least one alphabetic character.

Parameters

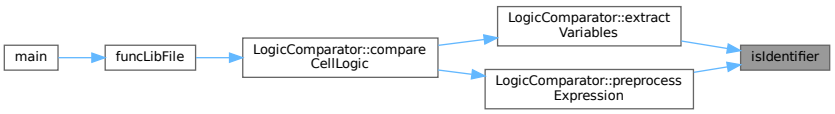
token	The string to check.

Returns

True if the string is a valid identifier, false otherwise.

Definition at line 74 of file LogicComparator.cpp.

Here is the caller graph for this function:



7.39.1.2 isOperator()

```
bool isOperator (
                const std::string & token )
```

Checks if a given token is a logical operator.

This function determines whether the input string `token` is one of the supported logical operators: "and", "or", "xor", or "not".

Parameters

token	The string to check.

Returns

True if the token is a logical operator, false otherwise.

Definition at line 99 of file LogicComparator.cpp.

Here is the caller graph for this function:

7.40 LogicComparator.cpp

Go to the documentation of this file.

```
00001 #include "LogicComparator.hpp"
00002
00003 LogicComparator::LogicComparator(const std::map<std::string, std::string> &ref_outpin_map,
00004                                  const std::map<std::string, std::string> &comp_outpin_map,
00005                                  const std::string &cell_name)
00006     : ref_outpin_map_(ref_outpin_map), comp_outpin_map_(comp_outpin_map), cell_name_(cell_name) {}
00024 void LogicComparator::logic() {
00025   typedef exprtk::symbol_table<double> symbol_table_t;
00026   typedef exprtk::expression<double> expression_t;
00027   typedef exprtk::parser<double> parser_t;
00028
00029   const std::string expression_string = "not(A and B) or C";
00030
00031   symbol_table_t symbol_table;
00032   symbol_table.create_variable("A");
00033   symbol_table.create_variable("B");
00034   symbol_table.create_variable("C");
00035
00036   expression_t expression;
00037   expression.register_symbol_table(symbol_table);
```

```
00038
00039   parser_t parser;
00040   parser.compile(expression_string, expression);
00041
00042   printf(" # | A | B | C | %s\n"
00043          "---+---+---+---+-%s\n",
00044          expression_string.c_str(), std::string(expression_string.size(), '-').c_str());
00045
00046   for (int i = 0; i < 8; ++i) {
00047     symbol_table.get_variable("A")->ref() = double(i & 0x01 ? 1 : 0);
00048     symbol_table.get_variable("B")->ref() = double(i & 0x02 ? 1 : 0);
00049     symbol_table.get_variable("C")->ref() = double(i & 0x04 ? 1 : 0);
00050
00051     const int result = static_cast<int>(expression.value());
00052
00053     printf(" %d | %d | %d | %d | %d \n", i,
00054            static_cast<int>(symbol_table.get_variable("A")->value()),
00055            static_cast<int>(symbol_table.get_variable("B")->value()),
00056            static_cast<int>(symbol_table.get_variable("C")->value()), result);
00057   }
00058 }
00059
00060 // --- Preprocessing Function ---
00061
00074 bool isIdentifier(const std::string &token) {
00075   if (token.empty())
00076     return false;
00077   // Check if the token is a valid identifier (upper alpha+ numeric + underscore)
00078   bool has_alpha = false;
00079   for (char c : token) {
00080     if (std::isalpha(c)) {
00081       has_alpha = true;
00082     } else if (!std::isalnum(c) && c != '_') {
00083       return false; // Contains invalid char
00084     }
00085   }
00086   // Must start with an uppercase alphabetic character
00087   return has_alpha && std::isupper(token[0]);
00088 }
00089
00099 bool isOperator(const std::string &token) {
00100   static const std::set<std::string> operators = {"and", "or", "xor", "not"};
00101   return operators.count(token);
00102 }
00103
00128 std::string LogicComparator::preprocessExpression(const std::string &input_expr) {
00129   std::string processed = input_expr;
00130   spdlog::debug("Preprocessing raw expression: {}", processed);
00131
00132   // 0. Trim leading/trailing whitespace first
00133   processed = std::regex_replace(processed, std::regex("^\\s+|\\s+$"), "");
00134   if (processed.empty()) {
00135     spdlog::debug("Input expression is empty.");
00136     return ""; // Handle empty input early
00137   }
00138
00139   // --- Helper function: build log string ---
00140   auto build_log_string = [](const auto &container) {
00141     std::ostringstream oss;
00142     for (auto it = container.begin(); it != container.end(); ++it) {
00143       oss << *it;
00144       if (std::next(it) != container.end()) {
00145         oss << ", ";
00146       }
00147     }
00148     return oss.str();
00149   };
00150
00151   // 1. Handle '!' for NOT carefully BEFORE adding general spaces.
```

```
00152   // Replace logical NOT '!' with " not ", ensuring not to replace '!='.
00153   std::string temp_processed;
00154   temp_processed.reserve(processed.length() * 1.2); // Pre-allocate a bit more space
00155   for (size_t i = 0; i < processed.length(); ++i) {
00156     if (processed[i] == '!') {
00157       if (i + 1 < processed.length() && processed[i + 1] == '=') {
00158         temp_processed += "!="; // Keep inequality operator
00159         i++;                    // Skip the '='
00160       } else {
00161         temp_processed += " not "; // Replace logical NOT with spaced operator
00162       }
00163     } else {
00164       temp_processed += processed[i];
00165     }
00166   }
00167   processed = temp_processed;
00168   spdlog::trace("After '!' to 'not' conversion: {}", processed);
00169
00170   // 2. Add spaces around other operators and parentheses that need them
00171   processed = std::regex_replace(processed, std::regex("\\("), " ( ");
00172   processed = std::regex_replace(processed, std::regex("\\)"), " ) ");
00173   processed = std::regex_replace(processed, std::regex("\\+"), " + "); // OR
00174   processed = std::regex_replace(processed, std::regex("\\^"), " ^ "); // XOR
00175   processed = std::regex_replace(processed, std::regex("\\*"), " * "); // AND
00176   // Consolidate multiple spaces into one and trim again
00177   processed = std::regex_replace(processed, std::regex("\\s+"), " ");
00178   processed = std::regex_replace(processed, std::regex("^\\s+|\\s+$"), "");
00179   spdlog::trace("After adding spaces: {}", processed);
00180
00181   // 3. Replace symbolic operators with their keyword equivalents
00182   //    Note: 'not' is already handled.
00183   processed = std::regex_replace(processed, std::regex(" \\+ "), " or ");
00184   processed = std::regex_replace(processed, std::regex(" \\^ "), " xor ");
00185   processed = std::regex_replace(processed, std::regex(" \\* "), " and ");
00186   spdlog::trace("After operator keyword replacement: {}", processed);
00187
00188   // 4. Tokenize based on spaces
00189   std::stringstream ss(processed);
00190   // Read tokens skipping whitespace
00191   std::vector<std::string> tokens{std::istream_iterator<std::string>(ss),
00192                                   std::istream_iterator<std::string>()};
00193
00194   if (tokens.empty()) {
00195     spdlog::debug("No tokens found after tokenization.");
00196     return "";
00197   }
00198   spdlog::trace("Tokens after initial processing: [{}]", build_log_string(tokens));
00199
00200   // 5. Insert implied 'and'
00201   std::vector<std::string> tokens_with_implied_and;
00202   if (!tokens.empty()) {
00203     tokens_with_implied_and.push_back(tokens[0]);
00204     for (size_t i = 0; i < tokens.size() - 1; ++i) {
00205       const std::string &current_token = tokens[i];
00206       const std::string &next_token = tokens[i + 1];
00207
00208       // An operand ends if it's an identifier or a closing parenthesis.
00209       bool current_ends_operand = isIdentifier(current_token) || current_token == ")";
00210       // An operand starts if it's an identifier, an opening parenthesis, or 'not'.
00211       bool next_starts_operand =
00212           isIdentifier(next_token) || next_token == "(" || next_token == "not";
00213
00214       // We should insert 'and' if the current token ends an operand AND the next token starts one,
00215       // UNLESS the current token is already an operator/opening bracket OR the next token is an
00216       // operator/closing bracket. Explicit check: Don't insert 'and' if an operator already exists
00217       // between them.
00218       bool current_is_op_or_open =
00219           isOperator(current_token) || current_token == "not" || current_token == "(";
00220       bool next_is_op_or_close = isOperator(next_token) || next_token == "not" || next_token == ")";
```

```
00221
00222       if (current_ends_operand && next_starts_operand && !current_is_op_or_open &&
00223           !next_is_op_or_close) {
00224         spdlog::trace("Inserting implied 'and' between '{}' and '{}'", current_token, next_token);
00225         tokens_with_implied_and.push_back("and");
00226       }
00227
00228       tokens_with_implied_and.push_back(next_token); // Add the next token regardless
00229     }
00230   } else {
00231     // Handle case where initial tokenization yielded no tokens
00232     return "";
00233   }
00234
00235   spdlog::trace("Tokens after implied 'and': [{}]", build_log_string(tokens_with_implied_and));
00236
00237   // 6. Handle 'not Identifier' ONLY during reconstruction (NEW LOGIC)
00238   std::vector<std::string> final_tokens;
00239   for (size_t i = 0; i < tokens_with_implied_and.size(); ++i) {
00240     const std::string &current_token = tokens_with_implied_and[i];
00241
00242     if (current_token == "not") {
00243       // Check if the *next* token is an Identifier
00244       if (i + 1 < tokens_with_implied_and.size() && isIdentifier(tokens_with_implied_and[i + 1])) {
00245         // Found 'not Identifier', transform to 'not(Identifier)'
00246         final_tokens.push_back("not");
00247         final_tokens.push_back("(");
00248         final_tokens.push_back(tokens_with_implied_and[i + 1]); // The identifier
00249         final_tokens.push_back(")");
00250         i++; // Increment i to skip the identifier we just processed
00251       } else {
00252         // 'not' is followed by '(', another operator, or is at the end.
00253         // Add 'not' as is, let ExprTk parse 'not(...)' or handle errors.
00254         final_tokens.push_back("not");
00255       }
00256     } else {
00257       // Token is not 'not', just add it to the final list
00258       final_tokens.push_back(current_token);
00259     }
00260   }
00261   spdlog::trace("Tokens after 'not' parenthesis handling: [{}]", build_log_string(final_tokens));
00262
00263   // 7. Reconstruct the final expression string from tokens
00264   std::string final_expr;
00265   if (!final_tokens.empty()) {
00266     final_expr = final_tokens[0];
00267     for (size_t i = 1; i < final_tokens.size(); ++i) {
00268       const std::string &prev = final_tokens[i - 1];
00269       const std::string &curr = final_tokens[i];
00270
00271       // Add space smartly: No space after '(', no space before ')',
00272       // no space after 'not' if followed by '(', no space before ',' (in function args, if
00273       // applicable)
00274       bool add_space = true;
00275       if (prev == "(" || curr == ")" || curr == ",") {
00276         add_space = false;
00277       }
00278       if (prev == "not" && curr == "(") {
00279         add_space = false; // Already handled by 'not(...)' structure
00280       }
00281       // Potentially add more rules if needed for other operators/functions
00282
00283       if (add_space) {
00284         final_expr += " ";
00285       }
00286       final_expr += curr;
00287     }
00288   }
00289
```

```
00290   // 8. Final cleanup (consolidate any remaining multiple spaces and trim)
00291   final_expr = std::regex_replace(final_expr, std::regex("\\s+"), " ");
00292   final_expr = std::regex_replace(final_expr, std::regex("^\\s+|\\s+$"), "");
00293
00294   spdlog::debug("Preprocessed expression result: {}", final_expr);
00295   return final_expr;
00296 }
00297
00298 // --- Variable Extraction Helper ---
00327 bool LogicComparator::extractVariables(const std::string &expr1_raw, const std::string &expr2_raw,
00328                                        std::vector<std::string> &sorted_vars) {
00329   // Define the set of ExprTk keywords/function names to filter out (lowercase)
00330   // Even though isIdentifier checks for uppercase, keep this filter as a safety measure
00331   static const std::set<std::string> keywords = {"abs",
00332                                                  "acos",
00333                                                  "acosh",
00334                                                  "and",
00335                                                  "asin",
00336                                                  "asinh",
00337                                                  "assert",
00338                                                  "atan",
00339                                                  "atan2",
00340                                                  "atanh",
00341                                                  "avg",
00342                                                  "break",
00343                                                  "case",
00344                                                  "ceil",
00345                                                  "clamp",
00346                                                  "continue",
00347                                                  "cosh",
00348                                                  "cos",
00349                                                  "cot",
00350                                                  "csc",
00351                                                  "default",
00352                                                  "deg2grad",
00353                                                  "deg2rad",
00354                                                  "else",
00355                                                  "equal",
00356                                                  "erfc",
00357                                                  "erf",
00358                                                  "exp",
00359                                                  "expm1",
00360                                                  "false",
00361                                                  "floor",
00362                                                  "for",
00363                                                  "frac",
00364                                                  "grad2deg",
00365                                                  "hypot",
00366                                                  "iclamp",
00367                                                  "if",
00368                                                  "ilike",
00369                                                  "in",
00370                                                  "inrange",
00371                                                  /*"in",*/ "like",
00372                                                  "log",
00373                                                  "log10",
00374                                                  "log1p",
00375                                                  "log2",
00376                                                  "logn",
00377                                                  "mand",
00378                                                  "max",
00379                                                  "min",
00380                                                  "mod",
00381                                                  "mor",
00382                                                  "mul",
00383                                                  "nand",
00384                                                  "ncdf",
00385                                                  "nor",
00386                                                  "not",
```

```
00387                                           "not_equal",
00388                                           /*"not",*/ "null",
00389                                           "or",
00390                                           "pow",
00391                                           "rad2deg",
00392                                           "repeat",
00393                                           "return",
00394                                           "root",
00395                                           "roundn",
00396                                           "round",
00397                                           "sec",
00398                                           "sgn",
00399                                           "shl",
00400                                           "shr",
00401                                           "sinc",
00402                                           "sinh",
00403                                           "sin",
00404                                           "sqrt",
00405                                           "sum",
00406                                           "swap",
00407                                           "switch",
00408                                           "tanh",
00409                                           "tan",
00410                                           "true",
00411                                           "trunc",
00412                                           "until",
00413                                           "var",
00414                                           "while",
00415                                           "xnor",
00416                                           "xor"};
00417
00418   // Regular expression to find potential identifiers (unchanged)
00419   const std::regex identifier_regex("[a-zA-Z_][a-zA-Z0-9_]*");
00420
00421   // Store validated and filtered variable names
00422   std::set<std::string> actual_vars1_set;
00423   std::set<std::string> actual_vars2_set;
00424
00425   // --- Helper function: build log string ---
00426   auto build_log_string = [](const auto &container) {
00427     std::ostringstream oss;
00428     oss << "[";
00429     for (auto it = container.begin(); it != container.end(); ++it) {
00430       oss << *it;
00431       if (std::next(it) != container.end()) {
00432         oss << ", ";
00433       }
00434     }
00435     oss << "]";
00436     return oss.str();
00437   };
00438
00439   // --- Process the first expression ---
00440   spdlog::debug("Extracting and validating identifiers from raw expr1: {}", expr1_raw);
00441   auto words_begin1 = std::sregex_iterator(expr1_raw.begin(), expr1_raw.end(), identifier_regex);
00442   auto words_end1 = std::sregex_iterator();
00443   for (std::sregex_iterator i = words_begin1; i != words_end1; ++i) {
00444     std::string potential_var = i->str();
00445     spdlog::trace("Regex found in expr1: {}", potential_var);
00446
00447     // 1. Validate using isIdentifier (now checks for uppercase etc.)
00448     if (isIdentifier(potential_var)) {
00449       // 2. Check if it's a keyword (still use lowercase comparison as a precaution)
00450       std::string lower_var = potential_var;
00451       std::transform(lower_var.begin(), lower_var.end(), lower_var.begin(),
00452                      [](unsigned char c) { return std::tolower(c); });
00453
00454       if (keywords.find(lower_var) == keywords.end()) {
00455         actual_vars1_set.insert(potential_var); // Keep original case
```

```
00456         spdlog::trace("Kept variable from expr1: {}", potential_var);
00457       } else {
00458         spdlog::trace("Filtered keyword from expr1: {}", potential_var);
00459       }
00460     } else {
00461       spdlog::trace("Filtered invalid identifier from expr1: {}", potential_var);
00462     }
00463   }
00464   spdlog::debug("Validated variables found in expr1: {}", build_log_string(actual_vars1_set));
00465
00466   // --- Process the second expression ---
00467   spdlog::debug("Extracting and validating identifiers from raw expr2: {}", expr2_raw);
00468   auto words_begin2 = std::sregex_iterator(expr2_raw.begin(), expr2_raw.end(), identifier_regex);
00469   auto words_end2 = std::sregex_iterator();
00470   for (std::sregex_iterator i = words_begin2; i != words_end2; ++i) {
00471     std::string potential_var = i->str();
00472     spdlog::trace("Regex found in expr2: {}", potential_var);
00473
00474     // 1. Validate using isIdentifier
00475     if (isIdentifier(potential_var)) {
00476       // 2. Check if it's a keyword (still use lowercase comparison)
00477       std::string lower_var = potential_var;
00478       std::transform(lower_var.begin(), lower_var.end(), lower_var.begin(),
00479                      [](unsigned char c) { return std::tolower(c); });
00480
00481       if (keywords.find(lower_var) == keywords.end()) {
00482         actual_vars2_set.insert(potential_var); // Keep original case
00483         spdlog::trace("Kept variable from expr2: {}", potential_var);
00484       } else {
00485         spdlog::trace("Filtered keyword from expr2: {}", potential_var);
00486       }
00487     } else {
00488       spdlog::trace("Filtered invalid identifier from expr2: {}", potential_var);
00489     }
00490   }
00491   spdlog::debug("Validated variables found in expr2: {}", build_log_string(actual_vars2_set));
00492
00493   // --- Compare the two variable sets ---
00494   if (actual_vars1_set == actual_vars2_set) {
00495     // Sets are equal, extract the variable list
00496     sorted_vars.assign(actual_vars1_set.begin(), actual_vars1_set.end());
00497     std::sort(sorted_vars.begin(), sorted_vars.end()); // Sort alphabetically
00498     spdlog::info("Variable sets match. Found unique variables for comparison: {}",
00499                  build_log_string(sorted_vars));
00500     spdlog::info("Extracted variables done !");
00501     return true; // Variable sets are consistent
00502   } else {
00503     // Sets are not equal, report an error and return false
00504     spdlog::warn("Variable sets do not match between the two expressions!");
00505     // Calculate the differences for more detailed logging
00506     std::vector<std::string> diff1, diff2;
00507     std::set_difference(actual_vars1_set.begin(), actual_vars1_set.end(), actual_vars2_set.begin(),
00508                         actual_vars2_set.end(),
00509                         std::back_inserter(diff1)); // Vars only in expr1
00510     std::set_difference(actual_vars2_set.begin(), actual_vars2_set.end(), actual_vars1_set.begin(),
00511                         actual_vars1_set.end(),
00512                         std::back_inserter(diff2)); // Vars only in expr2
00513
00514     if (!diff1.empty()) {
00515       spdlog::warn("Variables only in reference expression: {}", build_log_string(diff1));
00516     }
00517     if (!diff2.empty()) {
00518       spdlog::warn("Variables only in comparison expression: {}", build_log_string(diff2));
00519     }
00520     sorted_vars.clear(); // Clear the output variable list
00521     return false;        // Variable sets are inconsistent
00522   }
00523 }
00524
```

```
00525 // --- Implementation of compareSingleExpressionPair ---
00553 void LogicComparator::compareSingleExpressionPair(const std::string &ref_expression_processed,
00554                                                   const std::string &comp_expression_processed,
00555                                                   const std::vector<std::string> &sorted_vars,
00556                                                   PinComparisonResult &result) {
00557   // Store processed expressions in the result struct
00558   result.ref_expr_processed = ref_expression_processed;
00559   result.comp_expr_processed = comp_expression_processed;
00560   result.comparison_possible = true; // Assume comparison is possible initially
00561
00562   spdlog::debug("Comparing processed expressions for pin '{}':", result.pin_name);
00563   spdlog::debug("  Ref : {}", ref_expression_processed);
00564   spdlog::debug("  Comp: {}", comp_expression_processed);
00565
00566   // --- ExprTk Setup ---
00567   typedef exprtk::symbol_table<double> symbol_table_t;
00568   typedef exprtk::expression<double> expression_t;
00569   typedef exprtk::parser<double> parser_t;
00570
00571   // Setup for Reference Expression
00572   symbol_table_t ref_symbol_table;
00573   expression_t ref_expression;
00574   parser_t ref_parser;
00575
00576   for (const auto &var_name : sorted_vars) {
00577     // Create variable, ExprTk manages its memory within the table
00578     if (!ref_symbol_table.create_variable(var_name)) {
00579       result.error_message = "Failed to create variable '" + var_name + "' in ref symbol table.";
00580       spdlog::error(result.error_message);
00581       result.comparison_possible = false;
00582       return; // Cannot proceed
00583     }
00584   }
00585   ref_expression.register_symbol_table(ref_symbol_table);
00586
00587   // Compile Reference Expression
00588   result.ref_compiles = ref_parser.compile(ref_expression_processed, ref_expression);
00589   if (!result.ref_compiles) {
00590     result.error_message = "Reference expression compilation failed: " + ref_parser.error();
00591     spdlog::warn("Pin '{}': {}", result.pin_name, result.error_message);
00592     result.comparison_possible = false;
00593     // Don't return yet, maybe the comparison one also fails
00594   } else {
00595     spdlog::debug("Pin '{}': Reference expression compiled successfully.", result.pin_name);
00596   }
00597
00598   // Setup for Comparison Expression
00599   symbol_table_t comp_symbol_table;
00600   expression_t comp_expression;
00601   parser_t comp_parser;
00602
00603   for (const auto &var_name : sorted_vars) {
00604     if (!comp_symbol_table.create_variable(var_name)) {
00605       // Append error if ref also failed, otherwise set it
00606       std::string comp_err = "Failed to create variable '" + var_name + "' in comp symbol table.";
00607       result.error_message += (result.error_message.empty() ? "" : "\n") + comp_err;
00608       spdlog::error(comp_err);
00609       result.comparison_possible = false; // Mark as impossible now
00610       return;                             // Cannot proceed if variable creation fails
00611     }
00612   }
00613   comp_expression.register_symbol_table(comp_symbol_table);
00614
00615   // Compile Comparison Expression
00616   result.comp_compiles = comp_parser.compile(comp_expression_processed, comp_expression);
00617   if (!result.comp_compiles) {
00618     std::string comp_err = "Comparison expression compilation failed: " + comp_parser.error();
00619     result.error_message += (result.error_message.empty() ? "" : "\n") + comp_err;
00620     spdlog::warn("Pin '{}': {}", result.pin_name, comp_err);
```

```
00621     result.comparison_possible = false; // Mark as impossible if not already
00622   } else {
00623     spdlog::debug("Pin '{}': Comparison expression compiled successfully.", result.pin_name);
00624   }
00625
00626   // If either failed to compile, comparison is not possible
00627   if (!result.ref_compiles || !result.comp_compiles) {
00628     result.comparison_possible = false;
00629     return;
00630   }
00631
00632   // --- Truth Table Generation and Comparison ---
00633   size_t num_vars = sorted_vars.size();
00634   // Use unsigned long long for potentially large number of combinations
00635   unsigned long long num_combinations = 1ULL << num_vars;
00636   // Prevent excessively large tables (e.g., > 20 variables is 1M+ rows)
00637   const unsigned long long MAX_COMBINATIONS = 1ULL << 20; // Limit to 2^20 combinations
00638   if (num_vars > 20) {                                    // Check against var count for clarity
00639     result.error_message = "Too many variables (" + std::to_string(num_vars) +
00640                            "). Max supported for truth table is 20.";
00641     spdlog::error("Pin '{}': {}", result.pin_name, result.error_message);
00642     result.comparison_possible = false;
00643     return;
00644   }
00645   if (num_combinations == 0 && num_vars > 0) { // Overflow check
00646     result.error_message = "Number of combinations calculation overflowed for " +
00647                            std::to_string(num_vars) + " variables.";
00648     spdlog::error("Pin '{}': {}", result.pin_name, result.error_message);
00649     result.comparison_possible = false;
00650     return;
00651   }
00652
00653   std::vector<bool> ref_results;
00654   std::vector<bool> comp_results;
00655   ref_results.reserve(static_cast<size_t>(num_combinations)); // Avoid reallocations
00656   comp_results.reserve(static_cast<size_t>(num_combinations));
00657
00658   // --- Prepare Tabulate Tables ---
00659   Table ref_table;
00660   Table comp_table;
00661   Table::Row_t header_row;
00662   header_row.push_back("#"); // Row number column
00663   for (const auto &var_name : sorted_vars) {
00664     header_row.push_back(var_name);
00665   }
00666   header_row.push_back(ref_expression_processed); // Reference expression as last column header
00667   ref_table.add_row(header_row);
00668
00669   // Adjust header for comparison table
00670   header_row.back() = comp_expression_processed; // Change last header to comparison expression
00671   comp_table.add_row(header_row);
00672
00673   // Format header rows
00674   for (size_t i = 0; i < ref_table[0].size(); ++i) {
00675     ref_table[0][i].format().font_color(Color::yellow).font_style({FontStyle::bold});
00676     comp_table[0][i].format().font_color(Color::yellow).font_style({FontStyle::bold});
00677   }
00678
00679   spdlog::debug("Pin '{}': Generating truth table with {} variables ({} combinations)...",
00680                 result.pin_name, num_vars, num_combinations);
00681
00682   bool evaluation_error = false;
00683   for (unsigned long long i = 0; i < num_combinations; ++i) {
00684     Table::Row_t ref_data_row;
00685     Table::Row_t comp_data_row;
00686     ref_data_row.push_back(std::to_string(i));
00687     comp_data_row.push_back(std::to_string(i));
00688
00689     // Set variable values for this combination
```

```
00690     for (size_t j = 0; j < num_vars; ++j) {
00691       // The j-th bit of i determines the value of the j-th variable
00692       bool bit_value = ((i >> j) & 1ULL);
00693       double var_value = bit_value ? double(1.0) : double(0.0); // ExprTk uses floating point
00694 
00695       // Get variable reference and assign value
00696       // Need error checking here in theory, but create_variable should have caught issues
00697       ref_symbol_table.get_variable(sorted_vars[j])->ref() = var_value;
00698       comp_symbol_table.get_variable(sorted_vars[j])->ref() = var_value;
00699 
00700       // Add input value to table rows (as string "0" or "1")
00701       std::string bit_str = bit_value ? "1" : "0";
00702       ref_data_row.push_back(bit_str);
00703       comp_data_row.push_back(bit_str);
00704     }
00705 
00706     // Evaluate expressions
00707     double ref_val, comp_val;
00708     try {
00709       ref_val = ref_expression.value();
00710     } catch (const std::exception &e) {
00711       result.error_message +=
00712           "\nReference evaluation failed at combination " + std::to_string(i) + ": " + e.what();
00713       spdlog::error("Pin '{}': {}", result.pin_name, result.error_message);
00714       result.comparison_possible = false;
00715       evaluation_error = true;
00716       break; // Stop evaluation
00717     }
00718     try {
00719       comp_val = comp_expression.value();
00720     } catch (const std::exception &e) {
00721       result.error_message +=
00722           "\nComparison evaluation failed at combination " + std::to_string(i) + ": " + e.what();
00723       spdlog::error("Pin '{}': {}", result.pin_name, result.error_message);
00724       result.comparison_possible = false;
00725       evaluation_error = true;
00726       break; // Stop evaluation
00727     }
00728 
00729     // Store boolean results (commonly, non-zero is true in logic contexts)
00730     bool ref_bool_result = (ref_val != double(0.0));
00731     bool comp_bool_result = (comp_val != double(0.0));
00732 
00733     ref_results.push_back(ref_bool_result);
00734     comp_results.push_back(comp_bool_result);
00735 
00736     // Add output results to table rows (as string "0" or "1")
00737     ref_data_row.push_back(ref_bool_result ? "1" : "0");
00738     comp_data_row.push_back(comp_bool_result ? "1" : "0");
00739 
00740     // Add rows to tables
00741     ref_table.add_row(ref_data_row);
00742     comp_table.add_row(comp_data_row);
00743 
00744   } // End of combination loop
00745 
00746   if (evaluation_error) {
00747     return; // Don't proceed if evaluation failed
00748   }
00749 
00750   // --- Final Comparison ---
00751   if (result.comparison_possible) {
00752     result.are_equivalent = (ref_results == comp_results);
00753     if (result.are_equivalent) {
00754       spdlog::info("Pin '{}': Expressions ARE logically equivalent.", result.pin_name);
00755     } else {
00756       spdlog::warn("Pin '{}': Expressions ARE NOT logically equivalent.", result.pin_name);
00757       result.error_message +=
00758           "\nTruth table outputs differ."; // Add specific reason for non-equivalence
```

```
00759     }
00760
00761     // Store the generated tables in the result struct
00762     result.ref_truth_table = ref_table;
00763     result.comp_truth_table = comp_table;
00764   }
00765 }
00766
00812 void LogicComparator::compareCellLogic() {
00813
00814   // 1. Collect all unique pin names from both maps
00815   std::set<std::string> unique_pin_names;
00816   for (const auto &pair : ref_outpin_map_) {
00817     unique_pin_names.insert(pair.first);
00818   }
00819   for (const auto &pair : comp_outpin_map_) {
00820     unique_pin_names.insert(pair.first);
00821   }
00822
00823   spdlog::info("Starting logic comparison for cell '{}' with {} unique output pins...", cell_name_,
00824                unique_pin_names.size());
00825
00826   // 2. Iterate through each unique pin name
00827   for (const std::string &pin_name : unique_pin_names) {
00828     PinComparisonResult pin_result;
00829     pin_result.pin_name = pin_name;
00830     pin_result.comparison_possible = true; // Assume possible initially
00831
00832     // 3. Get raw expressions
00833     auto ref_it = ref_outpin_map_.find(pin_name);
00834     auto comp_it = comp_outpin_map_.find(pin_name);
00835
00836     if (ref_it == ref_outpin_map_.end()) {
00837       pin_result.error_message = "Pin not found in reference map.";
00838       spdlog::warn("Cell '{}', Pin '{}': {}", cell_name_, pin_name, pin_result.error_message);
00839       pin_result.comparison_possible = false;
00840       // Still try to get the comparison expression for reporting
00841       if (comp_it != comp_outpin_map_.end()) {
00842         pin_result.comp_expr_raw = comp_it->second;
00843       }
00844     } else {
00845       pin_result.ref_expr_raw = ref_it->second;
00846       spdlog::debug("Pin -> Expression: {} -> {}", pin_name, pin_result.ref_expr_raw);
00847     }
00848
00849     if (comp_it == comp_outpin_map_.end()) {
00850       std::string comp_err = "Pin not found in comparison map.";
00851       pin_result.error_message += (pin_result.error_message.empty() ? "" : "\n") + comp_err;
00852       spdlog::warn("Cell '{}', Pin '{}': {}", cell_name_, pin_name, comp_err);
00853       pin_result.comparison_possible = false;
00854       // If ref existed, store it
00855       if (ref_it != ref_outpin_map_.end()) {
00856         pin_result.ref_expr_raw = ref_it->second; // Already stored if ref exists
00857       }
00858     } else {
00859       pin_result.comp_expr_raw = comp_it->second;
00860       spdlog::debug("Pin => Expression: {} => {}", pin_name, pin_result.comp_expr_raw);
00861     }
00862
00863     // If pin missing in either, we can't compare, but store result and continue
00864     if (!pin_result.comparison_possible) {
00865       all_pin_results_[pin_name] = pin_result;
00866       continue;
00867     }
00868
00869     // 4. Preprocess expressions
00870     std::string ref_processed = preprocessExpression(pin_result.ref_expr_raw);
00871     std::string comp_processed = preprocessExpression(pin_result.comp_expr_raw);
00872
```

```
00873     if (ref_processed.empty()) {
00874       pin_result.error_message += "\nReference expression became empty after preprocessing.";
00875       spdlog::warn("Cell '{}', Pin '{}': {}", cell_name_, pin_name,
00876                    "Reference expression became empty after preprocessing.");
00877       pin_result.comparison_possible = false;
00878     }
00879     if (comp_processed.empty()) {
00880       std::string comp_err = "\nComparison expression became empty after preprocessing.";
00881       pin_result.error_message += (pin_result.error_message.empty() ? "" : "\n") + comp_err;
00882       spdlog::warn("Cell '{}', Pin '{}': {}", cell_name_, pin_name,
00883                    "Comparison expression became empty after preprocessing.");
00884       pin_result.comparison_possible = false; // Mark as impossible if not already
00885     }
00886     // If preprocessing failed for either, store and continue
00887     if (ref_processed.empty() || comp_processed.empty()) {
00888       pin_result.ref_expr_processed = ref_processed; // Store possibly empty strings
00889       pin_result.comp_expr_processed = comp_processed;
00890       all_pin_results_[pin_name] = pin_result;
00891       continue;
00892     }
00893
00894     // 5. Extract and validate variables
00895     std::vector<std::string> sorted_vars;
00896     // Pass RAW expressions to extractVariables as it uses regex on original format
00897     bool variables_match =
00898         extractVariables(pin_result.ref_expr_raw, pin_result.comp_expr_raw, sorted_vars);
00899
00900     if (!variables_match) {
00901       pin_result.error_message = "Variable sets do not match between expressions.";
00902       // extractVariables already logs details
00903       pin_result.comparison_possible = false;
00904       pin_result.ref_expr_processed = ref_processed; // Store processed even if vars mismatch
00905       pin_result.comp_expr_processed = comp_processed;
00906       all_pin_results_[pin_name] = pin_result;
00907       continue;
00908     }
00909     // --- Helper function: build log string ---
00910     auto build_log_string = [](const auto &container) {
00911       std::ostringstream oss;
00912       for (auto it = container.begin(); it != container.end(); ++it) {
00913         oss << *it;
00914         if (std::next(it) != container.end()) {
00915           oss << ", ";
00916         }
00917       }
00918       return oss.str();
00919     };
00920     spdlog::info("Pin '{}': Variable sets match. Found unique variables: {}", pin_name,
00921                  build_log_string(sorted_vars));
00922
00923     // 6. Compare the single pair using the processed expressions
00924     // Pass pin_result by reference - it will be populated by the function
00925     compareSingleExpressionPair(ref_processed, comp_processed, sorted_vars, pin_result);
00926
00927     // 7. Store the detailed result for this pin
00928     all_pin_results_[pin_name] = pin_result;
00929
00930   } // End of pin loop
00931
00932   spdlog::info("Logic comparison finished for cell '{}'.", cell_name_);
00933 }
00934
00974 void LogicComparator::generateReport(const std::string &output_file) {
00975   // --- 0. Start Info ---
00976   spdlog::info("Generating report for cell '{}' to '{}'...", cell_name_, output_file);
00977
00978   // --- 1. Determine Output Format ---
00979   bool output_markdown = false;
00980   try {
```

```
00981     // Use std::filesystem to reliably get the extension
00982     std::filesystem::path file_path(output_file);
00983     if (file_path.has_extension() && file_path.extension() == ".md") {
00984       output_markdown = true;
00985     }
00986   } catch (const std::exception &e) {
00987     spdlog::warn("Could not determine file extension for '{}': {}. Defaulting to plain text.",
00988                  output_file, e.what());
00989     output_markdown = false; // Default to plain text on error
00990   }
00991
00992   // --- 2. Open Output File ---
00993   std::ofstream outfile(output_file,
00994                         std::ios::app); // Use app to append if file exists, or create new
00995   if (!outfile.is_open()) {
00996     spdlog::error("Failed to open output report file: {}", output_file);
00997     return;
00998   }
00999   spdlog::info("Generating report for cell '{}' to '{}' (Format: {})...", cell_name_, output_file,
01000                output_markdown ? "Markdown" : "Plain Text");
01001
01002   // --- 3. Report Header ---
01003   outfile << "# Logic Equivalence Comparison Report\n\n";
01004   outfile << "**Cell Name: " << cell_name_ << "**\n\n";
01005   // Reference Pin Functions Table
01006   Table ref_pin_table;
01007   ref_pin_table.add_row({"Reference Pin", "Function"});
01008   ref_pin_table[0][0].format().font_style({FontStyle::bold});
01009   ref_pin_table[0][1].format().font_style({FontStyle::bold});
01010   for (const auto &[pin, function] : ref_outpin_map_) {
01011     ref_pin_table.add_row({pin, "`" + function + "`"}); // Add backticks to function
01012   }
01013   // Comparison Pin Functions Table
01014   Table comp_pin_table;
01015   comp_pin_table.add_row({"Comparison Pin", "Function"});
01016   comp_pin_table[0][0].format().font_style({FontStyle::bold});
01017   comp_pin_table[0][1].format().font_style({FontStyle::bold});
01018   for (const auto &[pin, function] : comp_outpin_map_) {
01019     comp_pin_table.add_row({pin, "`" + function + "`"}); // Add backticks to function
01020   }
01021
01022   // --- 4. Report Metadata ---
01023   outfile << "**Performed by " << APP_NAME << " v" << APP_VERSION << " from " << APP_AUTHOR;
01024   auto now = std::chrono::system_clock::now();
01025   std::time_t now_time_t = std::chrono::system_clock::to_time_t(now);
01026   std::tm *now_tm = std::localtime(&now_time_t);
01027   outfile << ". on: " << std::put_time(now_tm, "%c") << "**\n" << std::endl;
01028
01029   // --- 5. Legend Generation ---
01030   outfile << "## Legend\n\n";
01031   Table legend_table;
01032   legend_table.add_row({"Symbol", "Meaning"});
01033   legend_table[0][0].format().font_style({FontStyle::bold});
01034   legend_table[0][1].format().font_style({FontStyle::bold});
01035   legend_table.add_row({"[OK]", "Logically Equivalent"});
01036   legend_table.add_row({"[NO]", "Not Logically Equivalent"});
01037   legend_table.add_row({"[NA]", "Comparison Not Applicable (General)"}); // Changed from possible
01038   legend_table.add_row({"[VM]", "Variable Mismatch (Cannot Compare)"});
01039   legend_table.add_row({"[CE]", "Compilation Error (Cannot Compare)"});
01040   legend_table.add_row({"[EE]", "Evaluation Error (Cannot Compare)"});
01041   legend_table.add_row({"[PE]", "Preprocessing/Pin/Setup Error (Cannot Compare)"});
01042
01043   // --- 6. Export Legend Table ---
01044   if (output_markdown) {
01045     MarkdownExporter exporter;
01046     outfile << exporter.dump(ref_pin_table) << "\n";  // Export ref pin table
01047     outfile << exporter.dump(comp_pin_table) << "\n"; // Export comp pin table
01048     outfile << exporter.dump(legend_table) << "\n";   // Export legend table
01049   } else {
```

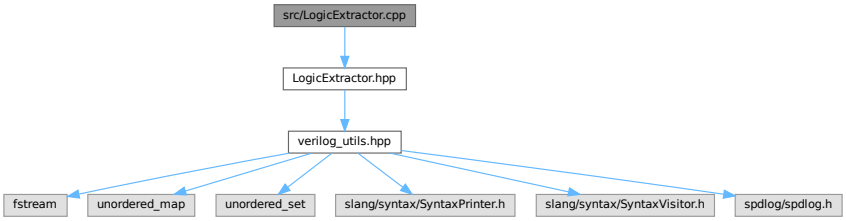
```
01050     outfile << ref_pin_table << "\n";  // Export ref pin table
01051     outfile << comp_pin_table << "\n"; // Export comp pin table
01052     outfile << legend_table << "\n";   // Export legend table (plain text)
01053   }
01054
01055   // --- 7. Process Pin Results ---
01056   for (const auto &[pin_name, result] : all_pin_results_) { // Use passed argument
01057     outfile << "## Pin: `" << pin_name << "`\n\n";          // Use backticks for pin name
01058
01059     // --- Output Truth Tables (Always if available) ---
01060     if (result.ref_truth_table.has_value()) {
01061       outfile << "### Reference Truth Table\n\n";
01062       Table ref_table = result.ref_truth_table.value();
01063       if (output_markdown) {
01064         MarkdownExporter exporter;
01065         outfile << exporter.dump(ref_table) << "\n";
01066       } else {
01067         outfile << ref_table << "\n";
01068       }
01069     } else {
01070       outfile << "*Reference truth table not generated (e.g., due to compilation error).*\n\n";
01071     }
01072
01073     if (result.comp_truth_table.has_value()) {
01074       outfile << "### Comparison Truth Table\n\n";
01075       Table comp_table = result.comp_truth_table.value();
01076       if (output_markdown) {
01077         MarkdownExporter exporter;
01078         outfile << exporter.dump(comp_table) << "\n";
01079       } else {
01080         outfile << comp_table << "\n";
01081       }
01082     } else {
01083       outfile << "*Comparison truth table not generated (e.g., due to compilation error).*\n\n";
01084     }
01085
01086     // --- Generate Summary Table ---
01087     outfile << "### Comparison Summary\n\n";
01088     Table summary_table;
01089     summary_table.add_row({"Property", "Value"});
01090     summary_table[0][0].format().font_style({FontStyle::bold});
01091     summary_table[0][1].format().font_style({FontStyle::bold});
01092     summary_table[0][0].format().font_color(Color::yellow);
01093     summary_table[0][1].format().font_color(Color::yellow);
01094
01095     // Determine Status Symbol using ASCII codes
01096     std::string status_symbol;
01097     if (!result.comparison_possible) {
01098       status_symbol = "[NA]"; // Default Not Applicable
01099       if (!result.error_message.empty()) {
01100         if (result.error_message.find("Variable sets do not match") != std::string::npos) {
01101           status_symbol = "[VM]";
01102         } else if (!result.ref_compiles || !result.comp_compiles ||
01103                    result.error_message.find("compilation failed") != std::string::npos) {
01104           // Check flags first, then error message as fallback
01105           status_symbol = "[CE]";
01106         } else if (result.error_message.find("evaluation failed") != std::string::npos) {
01107           status_symbol = "[EE]";
01108         } else {
01109           // If comparison_possible is false for other reasons (e.g., pin missing, setup errors)
01110           status_symbol = "[PE]";
01111         }
01112       } else if (!result.ref_compiles || !result.comp_compiles) {
01113         // Handle cases where comparison_possible might be true but compilation failed (should
01114         // ideally not happen if logic is sound)
01115         status_symbol = "[CE]";
01116       } else {
01117         status_symbol = "[PE]"; // Fallback if no error message but still not possible
01118       }
```

```
01119     } else if (result.are_equivalent) {
01120       status_symbol = "[OK]";
01121     } else {
01122       status_symbol = "[NO]";
01123     }
01124
01125     summary_table.add_row({"Status", status_symbol});
01126     summary_table.add_row({"Reference (Raw)", "`" + result.ref_expr_raw + "`"});
01127     summary_table.add_row({"Comparison (Raw)", "`" + result.comp_expr_raw + "`"});
01128     summary_table.add_row({"Reference (Processed)", "`" + result.ref_expr_processed + "`"});
01129     summary_table.add_row({"Comparison (Processed)", "`" + result.comp_expr_processed + "`"});
01130     summary_table.add_row({"Ref Expression Compiled", result.ref_compiles ? "Yes" : "No"});
01131     summary_table.add_row({"Comp Expression Compiled", result.comp_compiles ? "Yes" : "No"});
01132     summary_table.add_row({"Logically Equivalent", result.comparison_possible
01133                                                      ? (result.are_equivalent ? "Yes" : "No")
01134                                                      : "N/A"});
01135     summary_table[1][0].format().font_style({FontStyle::bold});
01136     summary_table[2][0].format().font_style({FontStyle::bold});
01137     summary_table[3][0].format().font_style({FontStyle::bold});
01138     summary_table[4][0].format().font_style({FontStyle::bold});
01139     summary_table[5][0].format().font_style({FontStyle::bold});
01140     summary_table[6][0].format().font_style({FontStyle::bold});
01141     summary_table[7][0].format().font_style({FontStyle::bold});
01142     summary_table[8][0].format().font_style({FontStyle::bold});
01143
01144     if (!result.error_message.empty()) {
01145       std::string formatted_error = result.error_message;
01146       if (output_markdown) {
01147         // Basic newline replacement for Markdown
01148         size_t pos = 0;
01149         while ((pos = formatted_error.find('\n', pos)) != std::string::npos) {
01150           formatted_error.replace(pos, 1, "<br>");
01151           pos += 4; // Length of "<br>"
01152         }
01153       }
01154       summary_table.add_row({"Details/Error", formatted_error});
01155     }
01156
01157     // Output Summary Table
01158     if (output_markdown) {
01159       MarkdownExporter exporter;
01160       outfile << exporter.dump(summary_table) << "\n";
01161     } else {
01162       outfile << summary_table << "\n";
01163     }
01164     std::cout << summary_table << "\n"; // Print to console for immediate feedback
01165
01166     outfile << "---\n\n"; // Separator between pins
01167
01168   } // End of pin results loop
01169
01170   outfile.close();
01171   spdlog::info("Report generation complete for cell '{}'.", cell_name_);
01172 }
```

7.41 src/LogicExtractor.cpp File Reference

```
#include "LogicExtractor.hpp"
```

Include dependency graph for LogicExtractor.cpp:



Functions

- void extractAndPrintNetlistInfo (const std::string &verilog_file, const std::string &cell)

  Extracts and prints netlist information from a Verilog file for a specified cell.

- std::map< std::string, std::string > extractLogicFromVerilog (const std::string &verilog_file, const std::string &cell)

  Extracts logic expressions from a Verilog file for a specified cell.

7.41.1 Function Documentation

7.41.1.1 extractAndPrintNetlistInfo()

```
void extractAndPrintNetlistInfo (
                const std::string & verilog_file,
                const std::string & cell )
```

Extracts and prints netlist information from a Verilog file for a specified cell.

This function parses a Verilog file using the slang library, extracts information about the primary inputs, primary outputs, internal wires, and gate drivers within a specified cell (module). It then prints a summary of the extracted information to the console using spdlog.

Parameters

verilog_file	The path to the Verilog file to be parsed.
cell	The name of the cell (module) for which to extract netlist information.

Exceptions

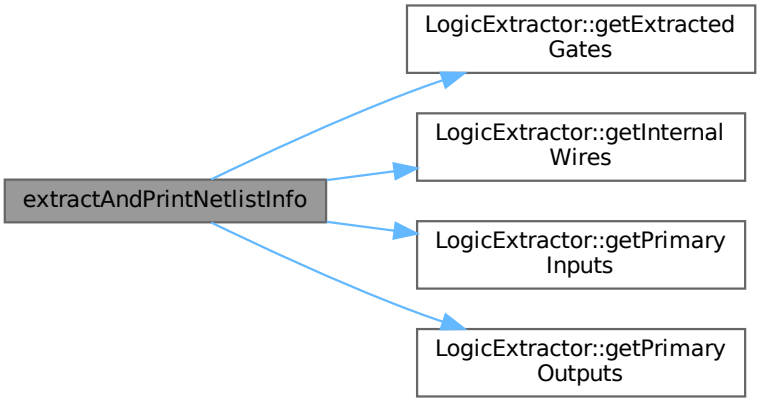
std::exception	If any error occurs during Verilog parsing or info extraction.

Note

The function uses the slang library for Verilog parsing and a custom LogicExtractor class to extract the desired information. Error messages are logged using spdlog.

Definition at line 676 of file LogicExtractor.cpp.

Here is the call graph for this function:



7.41.1.2 extractLogicFromVerilog()

```
std::map< std::string, std::string > extractLogicFromVerilog (
                const std::string & verilog_file,
                const std::string & cell )
```

Extracts logic expressions from a Verilog file for a specified cell.

This function parses a Verilog file using the slang library, identifies the specified cell, and extracts the logic expressions for its outputs. It returns a map where the keys are output signal names and the values are their corresponding logic expressions as strings.

Parameters

verilog_file	The path to the Verilog file to parse.
cell	The name of the cell (module) for which to extract logic expressions.

Returns

A map of output signal names to their logic expressions. Returns an empty map if parsing fails, the cell is not found, or no logic expressions can be derived.

Note

The function uses the slang library for Verilog parsing. Ensure that slang is properly installed and configured before using this function.

The logic extraction process involves traversing the syntax tree of the Verilog code and identifying relevant assignments and expressions within the specified cell.

Error messages and warnings are logged using the spdlog library.

Definition at line 755 of file LogicExtractor.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

7.42 LogicExtractor.cpp

Go to the documentation of this file.

```
00001 #include "LogicExtractor.hpp"
00002
00003 // --- Implementation for LogicExtractor ---
```

```
00004
00018 void LogicExtractor::handle(const slang::syntax::ModuleDeclarationSyntax &module) {
00019   if (parsingComplete_)
00020     return; // Don't re-process if called again
00021
00022   if (module.header && module.header->name.valid()) {
00023     std::string_view moduleName = module.header->name.valueText();
00024     if (!targetCell_.empty() && moduleName == targetCell_) {
00025       spdlog::info("LogicExtractor: Found target module: {}", targetCell_);
00026       inTargetModule_ = true;
00027
00028       // Reset state for the target module
00029       primaryInputs_.clear();
00030       primaryOutputs_.clear();
00031       internalWires_.clear();
00032       portDirections_.clear(); // Clear temporary direction map
00033       gateOutputDrivers_.clear();
00034       // logicCache_.clear(); // For Step 2
00035
00036       // Visit children (ports, declarations, instances) IN ORDER
00037       // It's often better to visit declarations first, then the port list
00038       // Slang's default visitDefault might handle this, but be aware.
00039       visitDefault(module);
00040
00041       // --- Finalize Ports after visiting declarations and port list ---
00042       // This part might be better placed *after* visiting NonAnsiPortList,
00043       // assuming the list visit confirms the names from the header.
00044       // Let's move the finalization logic to handle(NonAnsiPortListSyntax)
00045
00046       // Mark that we finished processing the target module
00047       inTargetModule_ = false; // Important to prevent processing other modules
00048       parsingComplete_ = true; // Stop processing after finding the target
00049       spdlog::info("LogicExtractor: Finished visiting target module '{}'.", targetCell_);
00050
00051     } else if (!parsingComplete_) {
00052       // If not the target module yet, continue searching
00053       visitDefault(module);
00054     }
00055   } else if (!parsingComplete_) {
00056     // Handle modules without valid headers if necessary, or just traverse
00057     visitDefault(module);
00058   }
00059 }
00060
00061 // Handle Port Declarations (defines direction and type, usually inside module body)
00092 void LogicExtractor::handle(const slang::syntax::PortDeclarationSyntax &portDecl) {
00093   if (!inTargetModule_)
00094     return;
00095   spdlog::debug("LogicExtractor: Handling Port Declaration");
00096
00097   std::string direction = "unknown";
00098   if (portDecl.header) {
00099     // Determine direction (input/output/inout)
00100     // Important: Use valueText() as Slang typically represents keywords as text tokens
00101     if (auto varHeader = portDecl.header->as_if<slang::syntax::VariablePortHeaderSyntax>()) {
00102       if (varHeader->direction.valid()) {
00103         direction = std::string(varHeader->direction.valueText());
00104       }
00105     } else if (auto netHeader = portDecl.header->as_if<slang::syntax::NetPortHeaderSyntax>()) {
00106       if (netHeader->direction.valid()) {
00107         direction = std::string(netHeader->direction.valueText());
00108       }
00109     }
00110     // Add InterfacePortHeaderSyntax if needed
00111   } else {
00112     spdlog::warn("LogicExtractor: Port declaration without header found.");
00113   }
00114
00115   for (const auto &declarator : portDecl.declarators) {
```

```
00116     if (declarator->name.valid()) {
00117       std::string portName = std::string(declarator->name.valueText());
00118       if (portDirections_.count(portName) && portDirections_[portName] != "unknown") {
00119         spdlog::warn("LogicExtractor: Multiple direction declarations for port '{}'. Keeping first "
00120                      "one ('{}'). New direction: '{}'",
00121                      portName, portDirections_[portName], direction);
00122       } else {
00123         spdlog::info("LogicExtractor: Storing direction '{}' for port '{}'", direction, portName);
00124         portDirections_[portName] = direction;
00125         if (direction == "input") {
00126           primaryInputs_.insert(portName);
00127           internalWires_.insert(portName); // Inputs are also technically 'wires' usable internally
00128         } else if (direction == "output") {
00129           primaryOutputs_.insert(portName);
00130           internalWires_.insert(portName); // Outputs are also wires driven by something
00131         } else if (direction == "inout") {
00132           spdlog::info("LogicExtractor: Inout port '{}' found. Adding to both inputs and outputs.",
00133                        portName);
00134           primaryInputs_.insert(portName);
00135           primaryOutputs_.insert(portName);
00136           internalWires_.insert(portName);
00137         } else {
00138           spdlog::warn("LogicExtractor: Port '{}' found in declaration but has unknown or missing "
00139                        "direction '{}'. Treating as internal wire.",
00140                        portName, direction);
00141           internalWires_.insert(portName);
00142         }
00143         // logicCache_[portName] = portName; // For Step 2
00144       }
00145     } else {
00146       spdlog::warn(
00147           "LogicExtractor: Port declarator without a name found in PortDeclarationSyntax.");
00148     }
00149   }
00150   // Don't call visitDefault here, as it might revisit things unexpectedly.
00151   // Let the main module visitor handle traversing into children if necessary.
00152 }
00153
00154 // Handle Non-ANSI Port List (defines names, usually in module header)
00179 void LogicExtractor::handle(const slang::syntax::NonAnsiPortListSyntax &portList) {
00180   if (!inTargetModule_)
00181     return;
00182   spdlog::debug("LogicExtractor: Handling NonAnsi Port List");
00183
00184   for (const auto portSyntax : portList.ports) {
00185     if (!portSyntax)
00186       continue;
00187
00188     std::string portName = "";
00189     // Most common case: ImplicitNonAnsiPortSyntax contains the expression (usually just the name)
00190     if (auto implicitPort = portSyntax->as_if<slang::syntax::ImplicitNonAnsiPortSyntax>()) {
00191       if (implicitPort->expr) {
00192         // The expression itself might be complex, try to get the simple name
00193         if (auto portRef = implicitPort->expr->as_if<slang::syntax::PortReferenceSyntax>()) {
00194           if (portRef->name.valid()) {
00195             portName = std::string(portRef->name.valueText());
00196           }
00197         } else if (auto portConcat =
00198                        implicitPort->expr->as_if<slang::syntax::PortConcatenationSyntax>()) {
00199           // Handle concatenations if necessary - complex
00200           spdlog::warn("LogicExtractor: Port concatenation found in NonAnsi port list - currently "
00201                        "not fully handled for logic extraction. Port: {}",
00202                        implicitPort->toString());
00203           continue; // Skip complex ports for now
00204         } else {
00205           // Fallback: try getting name from the expression directly (might be just identifier)
00206           portName = implicitPort->toString();
00207         }
00208       }
```

```
00209     }
00210     // Handle ExplicitNonAnsiPortSyntax (e.g., .A(A)) if needed
00211     else if (auto explicitPort = portSyntax->as_if<slang::syntax::ExplicitNonAnsiPortSyntax>()) {
00212       if (explicitPort->name.valid()) {
00213         portName = std::string(explicitPort->name.valueText());
00214         // Note: explicitPort->expr is the internal signal it connects to.
00215         // We are primarily interested in the port's own name (explicitPort->name) here.
00216       }
00217     }
00218     // Handle EmptyNonAnsiPortSyntax (commas for placeholders) if necessary
00219     else if (portSyntax->kind == slang::syntax::SyntaxKind::EmptyNonAnsiPort) {
00220       spdlog::debug("LogicExtractor: Skipping empty non-ANSI port placeholder.");
00221       continue;
00222     } else {
00223       spdlog::warn("LogicExtractor: Unhandled NonAnsi port syntax kind: {}",
00224                    slang::syntax::toString(portSyntax->kind));
00225       continue;
00226     }
00227
00228     // Now we (hopefully) have the portName.
00229     if (!portName.empty()) {
00230       if (portDirections_.count(portName)) {
00231         const std::string &direction = portDirections_[portName];
00232         spdlog::debug("LogicExtractor: Finalizing port '{}' with direction '{}'", portName,
00233                       direction);
00234
00235         if (direction == "input") {
00236           primaryInputs_.insert(portName);
00237           internalWires_.insert(portName); // Inputs are also technically 'wires' usable internally
00238                                            // logicCache_[portName] = portName; // For Step 2
00239         } else if (direction == "output") {
00240           primaryOutputs_.insert(portName);
00241           internalWires_.insert(portName); // Outputs are also wires driven by something
00242         } else if (direction == "inout") {
00243           spdlog::info("LogicExtractor: Inout port '{}' found. Adding to both inputs and outputs.",
00244                        portName);
00245           primaryInputs_.insert(portName);
00246           primaryOutputs_.insert(portName);
00247           internalWires_.insert(portName);
00248           // logicCache_[portName] = portName; // For Step 2
00249         } else {
00250           spdlog::warn("LogicExtractor: Port '{}' found in list but has unknown or missing "
00251                        "direction '{}'. Treating as internal wire.",
00252                        portName, direction);
00253           internalWires_.insert(portName);
00254         }
00255       } else {
00256         spdlog::debug("LogicExtractor: Port '{}' found in NonAnsi list, no direction declaration ",
00257                       portName);
00258       }
00259     } else {
00260       spdlog::warn("LogicExtractor: Could not determine port name from NonAnsi port list item: {}",
00261                    portSyntax->toString());
00262     }
00263   }
00264   // Don't call visitDefault here
00265 }
00266
00267 // Handle explicit wire declarations
00277 void LogicExtractor::handle(const slang::syntax::NetDeclarationSyntax &netDecl) {
00278   if (!inTargetModule_)
00279     return;
00280
00281   for (const auto &declarator : netDecl.declarators) {
00282     if (declarator->name.valid()) {
00283       std::string wireName = std::string(declarator->name.valueText());
00284       spdlog::debug("LogicExtractor: Found wire declaration: {}", wireName);
00285       // Avoid adding ports again if they were also declared as nets (common)
00286       if (!primaryInputs_.count(wireName) && !primaryOutputs_.count(wireName)) {
```

```
00287         internalWires_.insert(wireName);
00288       } else {
00289         spdlog::trace("LogicExtractor: Wire '{}' is already known as a port.", wireName);
00290       }
00291     }
00292   }
00293   // Don't call visitDefault here
00294 }
00295
00296 // Handle primitive gate instantiations (MOST IMPORTANT PART)
00309 void LogicExtractor::handle(const slang::syntax::PrimitiveInstantiationSyntax &primitiveInst) {
00310   if (!inTargetModule_)
00311     return;
00312
00313   if (!primitiveInst.type.valid()) {
00314     spdlog::warn("LogicExtractor: Primitive instance without a type token found.");
00315     return;
00316   }
00317
00318   // Use token kind for reliable checking, text for name storage
00319   slang::parsing::TokenKind gateKind = primitiveInst.type.kind;
00320   std::string gateTypeName = std::string(primitiveInst.type.valueText());
00321
00322   spdlog::debug("LogicExtractor: Found Primitive Instance of Type: {} (Kind: {})", gateTypeName,
00323                 slang::parsing::toString(gateKind));
00324
00325   for (const auto &instance : primitiveInst.instances) {
00326     // if (!instance || !instance->connections.isInitialized()) { // Check if connections are valid
00327     //   spdlog::warn("LogicExtractor: Skipping primitive instance of type {} due to null pointer or
00328     //   "
00329     //                "uninitialized connections.",
00330     //                gateTypeName);
00331     //   continue;
00332     // }
00333
00334     GateInfo currentGateInfo;
00335     currentGateInfo.gateTypeName = gateTypeName;
00336     currentGateInfo.kind = gateKind;
00337
00338     // Primitives usually have ordered connections.
00339     // The FIRST connection is typically the OUTPUT.
00340     // The REST are INPUTS.
00341     if (instance->connections.empty()) {
00342       spdlog::warn("LogicExtractor: Gate instance of type {} has no connections.", gateTypeName);
00343       continue;
00344     }
00345
00346     // Extract Output Signal
00347     // Ensure the connection itself is not null
00348     if (!instance->connections[0]) {
00349       spdlog::error("LogicExtractor: First connection (output) is null for primitive {}",
00350                     gateTypeName);
00351       continue;
00352     }
00353     if (auto firstConn =
00354             instance->connections[0]->as_if<slang::syntax::OrderedPortConnectionSyntax>()) {
00355       currentGateInfo.outputSignal =
00356           firstConn->expr->toString(); // Get the output signal name from the connection
00357                                        // remove extra spaces in the signal name
00358       currentGateInfo.outputSignal.erase(
00359           std::remove_if(currentGateInfo.outputSignal.begin(), currentGateInfo.outputSignal.end(),
00360                          [](unsigned char x) { return std::isspace(x); }),
00361           currentGateInfo.outputSignal.end());
00362       if (!currentGateInfo.outputSignal.empty()) {
00363         spdlog::debug("  Output Signal: {}", currentGateInfo.outputSignal);
00364         // Gate outputs are internal signals (unless they are module outputs)
00365         // Add to internal wires if not already a primary output.
00366         if (!primaryOutputs_.count(currentGateInfo.outputSignal)) {
00367           internalWires_.insert(currentGateInfo.outputSignal);
```

```
00368         }
00369       } else {
00370         spdlog::error(
00371             "LogicExtractor: Could not determine output signal name for gate instance of type {}",
00372             gateTypeName);
00373         continue; // Skip this instance if output is unknown
00374       }
00375     } else {
00376       spdlog::error("LogicExtractor: Expected OrderedPortConnectionSyntax for output of primitive "
00377                     "{}, but got kind: {}. Conn: {}",
00378                     gateTypeName, slang::syntax::toString(instance->connections[0]->kind),
00379                     instance->connections[0]->toString());
00380       continue; // Skip this instance
00381     }
00382
00383     // Extract Input Signals
00384     for (size_t i = 1; i < instance->connections.size(); ++i) {
00385       // Ensure the connection itself is not null
00386       if (!instance->connections[i]) {
00387         spdlog::warn("LogicExtractor: Input connection {} is null for primitive {}", i,
00388                      gateTypeName);
00389         continue;
00390       }
00391       if (auto conn =
00392               instance->connections[i]->as_if<slang::syntax::OrderedPortConnectionSyntax>()) {
00393         std::string inputSig =
00394             conn->expr->toString(); // Get the input signal name from the connection
00395         // remove extra spaces in the signal name
00396         inputSig.erase(std::remove_if(inputSig.begin(), inputSig.end(),
00397                                       [](unsigned char x) { return std::isspace(x); }),
00398                        inputSig.end());
00399         if (!inputSig.empty()) {
00400           currentGateInfo.inputSignals.push_back(inputSig);
00401           spdlog::debug("  Input Signal {}: {}", i, inputSig);
00402         } else {
00403           spdlog::warn("LogicExtractor: Could not determine input signal name for input {} of gate "
00404                        "instance type {}. Conn: {}",
00405                        i, gateTypeName, conn->toString());
00406           // Decide whether to skip or continue with partial inputs
00407         }
00408       } else {
00409         spdlog::warn("LogicExtractor: Expected OrderedPortConnectionSyntax for input {} of "
00410                      "primitive {}, but got kind: {}. Conn: {}",
00411                      i, gateTypeName, slang::syntax::toString(instance->connections[i]->kind),
00412                      instance->connections[i]->toString());
00413       }
00414     }
00415
00416     // Store the gate information, mapping the output signal to its driving gate
00417     if (!currentGateInfo.outputSignal.empty()) {
00418       if (gateOutputDrivers_.count(currentGateInfo.outputSignal)) {
00419         // This is a critical warning - indicates multiple drivers for the same net!
00420         spdlog::error("LogicExtractor: Multiple drivers found for signal '{}'! Previous driver: "
00421                       "{}, New driver: {}. Netlist is likely invalid.",
00422                       currentGateInfo.outputSignal,
00423                       gateOutputDrivers_[currentGateInfo.outputSignal].gateTypeName, gateTypeName);
00424         // Keep the first one found for now, or decide on error handling
00425       } else {
00426         spdlog::info("  Storing driver for '{}': Gate Type '{}'", currentGateInfo.outputSignal,
00427                      gateTypeName);
00428         gateOutputDrivers_[currentGateInfo.outputSignal] = currentGateInfo;
00429       }
00430     }
00431   }
00432   // Don't call visitDefault here
00433 }
00434
00435 // --- Logic Derivation Implementation ---
00436
```

```
00437 // Public method called after visiting the tree
00451 std::map<std::string, std::string> LogicExtractor::getLogicExpressions() {
00452   std::map<std::string, std::string> result_map;
00453   if (!parsingComplete_) {
00454     spdlog::error("LogicExtractor: AST parsing did not complete or target module '{}' not found. "
00455                   "Cannot extract logic.",
00456                   targetCell_);
00457     return result_map; // Return empty map
00458   }
00459
00460   spdlog::info("LogicExtractor: Deriving logic expressions for {} output ports...",
00461                primaryOutputs_.size());
00462
00463   for (const std::string &outputPort : primaryOutputs_) {
00464     spdlog::debug("LogicExtractor: Deriving logic for output: {}", outputPort);
00465     try {
00466       result_map[outputPort] = deriveLogicRecursive(outputPort);
00467       spdlog::info("  Output: {} => {}", outputPort, result_map[outputPort]);
00468     } catch (const std::runtime_error &e) {
00469       spdlog::error("LogicExtractor: Error deriving logic for output '{}': {}", outputPort,
00470                     e.what());
00471       result_map[outputPort] = "/* Error deriving logic */";
00472     } catch (...) {
00473       spdlog::error("LogicExtractor: Unknown error deriving logic for output '{}'", outputPort);
00474       result_map[outputPort] = "/* Unknown error deriving logic */";
00475     }
00476   }
00477   return result_map;
00478 }
00479
00480 // Recursive function with memoization
00510 std::string LogicExtractor::deriveLogicRecursive(const std::string &signalName) {
00511   // 1. Check Cache (Memoization)
00512   if (logicCache_.count(signalName)) {
00513     return logicCache_.at(signalName);
00514   }
00515
00516   // 2. Base Case: Is it a primary input?
00517   if (primaryInputs_.count(signalName)) {
00518     // Already cached during port handling, but double-check
00519     if (!logicCache_.count(signalName)) {
00520       logicCache_[signalName] = signalName;
00521     }
00522     return signalName;
00523   }
00524
00525   // 3. Recursive Step: Is it driven by a gate?
00526   if (gateOutputDrivers_.count(signalName)) {
00527     const GateInfo &driverGate = gateOutputDrivers_.at(signalName);
00528     std::vector<std::string> inputExpressions;
00529     spdlog::debug("    Tracing signal '{}', driven by {} gate", signalName,
00530                   driverGate.gateTypeName);
00531
00532     // Recursively find expressions for all inputs of this gate
00533     for (const std::string &inputSig : driverGate.inputSignals) {
00534       if (inputSig.empty()) {
00535         throw std::runtime_error("Empty input signal name encountered for gate driving " +
00536                                  signalName);
00537       }
00538       spdlog::debug("      Recursing for input: {}", inputSig);
00539       inputExpressions.push_back(deriveLogicRecursive(inputSig));
00540     }
00541
00542     // Format the expression based on gate type and input expressions
00543     std::string currentExpr = formatExpression(driverGate, inputExpressions);
00544
00545     // Cache the result
00546     logicCache_[signalName] = currentExpr;
00547     return currentExpr;
```

```
00548   }
00549
00550   // 4. Handle Assign statements (if implemented)
00551   // if (assignDrivers_.count(signalName)) { ... }
00552
00553   // 5. Error Case: Signal not found or not driven by known element
00554   // Check if it's just an internal wire that wasn't driven?
00555   if (internalWires_.count(signalName)) {
00556     throw std::runtime_error(
00557         "Signal '" + signalName +
00558         "' is an internal wire but has no identified driver (gate or assign).");
00559   } else {
00560     throw std::runtime_error(
00561         "Signal '" + signalName +
00562         "' is not a primary input, known wire, or driven by a recognized gate/assignment.");
00563   }
00564 }
00565
00566 // Helper to format the expression string based on gate type
00584 std::string LogicExtractor::formatExpression(const GateInfo &gateInfo,
00585                                              const std::vector<std::string> &inputExprs) {
00586   if (inputExprs.empty() && gateInfo.kind != slang::parsing::TokenKind::NotKeyword &&
00587       gateInfo.kind != slang::parsing::TokenKind::BufKeyword) {
00588     // Gates like AND/OR/XOR need inputs
00589     spdlog::warn("Gate type {} requires inputs, but none were provided/derived for output {}",
00590                  gateInfo.gateTypeName, gateInfo.outputSignal);
00591     return "/*<Error: Missing Inputs for " + gateInfo.gateTypeName + ">*/";
00592   }
00593
00594   std::string result = "";
00595
00596   // Use gateInfo.type (enum) for reliable checking
00597   switch (gateInfo.kind) {
00598   case slang::parsing::TokenKind::AndKeyword:
00599     result = "(" + inputExprs[0];
00600     for (size_t i = 1; i < inputExprs.size(); ++i)
00601       result += " * " + inputExprs[i]; // Use * for AND
00602     result += ")";
00603     break;
00604   case slang::parsing::TokenKind::NandKeyword:
00605     result = "!(" + inputExprs[0];
00606     for (size_t i = 1; i < inputExprs.size(); ++i)
00607       result += " * " + inputExprs[i];
00608     result += ")";
00609     break;
00610   case slang::parsing::TokenKind::OrKeyword:
00611     result = "(" + inputExprs[0];
00612     for (size_t i = 1; i < inputExprs.size(); ++i)
00613       result += " + " + inputExprs[i]; // Use + for OR
00614     result += ")";
00615     break;
00616   case slang::parsing::TokenKind::NorKeyword:
00617     result = "!(" + inputExprs[0];
00618     for (size_t i = 1; i < inputExprs.size(); ++i)
00619       result += " + " + inputExprs[i];
00620     result += ")";
00621     break;
00622   case slang::parsing::TokenKind::XorKeyword:
00623     result = "(" + inputExprs[0];
00624     for (size_t i = 1; i < inputExprs.size(); ++i)
00625       result += " ^ " + inputExprs[i]; // Use ^ for XOR
00626     result += ")";
00627     break;
00628   case slang::parsing::TokenKind::XnorKeyword:
00629     result = "!(" + inputExprs[0];
00630     for (size_t i = 1; i < inputExprs.size(); ++i)
00631       result += " ^ " + inputExprs[i];
00632     result += ")";
00633     break;
```

```
00634   case slang::parsing::TokenKind::NotKeyword: // NOT gate
00635     if (inputExprs.size() != 1) {
00636       spdlog::warn("NOT gate expects 1 input, got {}", inputExprs.size());
00637       return "/*<Error: Incorrect Inputs for NOT>*/";
00638     }
00639     result = "!" + inputExprs[0]; // Use ! for NOT
00640     break;
00641   case slang::parsing::TokenKind::BufKeyword: // BUF gate
00642     if (inputExprs.size() != 1) {
00643       spdlog::warn("BUF gate expects 1 input, got {}", inputExprs.size());
00644       return "/*<Error: Incorrect Inputs for BUF>*/";
00645     }
00646     result = inputExprs[0]; // Output is the same as input
00647     break;
00648   // Add cases for bufif0, bufif1, notif0, notif1, pullup, pulldown, cmos, nmos, pmos, tran etc. if
00649   // needed
00650   default:
00651     spdlog::warn("Unsupported primitive gate type for logic expression generation: {}",
00652                  gateInfo.gateTypeName);
00653     result = "/*<Unsupported Gate: " + gateInfo.gateTypeName + ">*/";
00654     break;
00655   }
00656   return result;
00657 }
00658
00659 // --- New Function Implementation (Step 1: Visit and Print) ---
00660
00676 void extractAndPrintNetlistInfo(const std::string &verilog_file, const std::string &cell) {
00677   spdlog::info("--- Step 1: Starting Netlist Info Extraction ---");
00678   spdlog::info("Verilog file: '{}', Target cell: '{}'", verilog_file, cell);
00679
00680   try {
00681     auto result = slang::syntax::SyntaxTree::fromFile(verilog_file);
00682     if (!result) { // Check if result is valid
00683       spdlog::error("Error parsing Verilog file '{}'. Cannot extract info.", verilog_file);
00684       return;
00685     }
00686
00687     spdlog::info("Successfully parsed Verilog file.");
00688     std::shared_ptr<slang::syntax::SyntaxTree> tree = result.value();
00689
00690     LogicExtractor extractor(cell);
00691     tree->root().visit(extractor); // Populate extractor's internal state
00692
00693     // --- Print Summary (Debug for Step 1) ---
00694     spdlog::info("--- Extraction Summary for Cell '{}': ---", cell);
00695
00696     const auto &inputs = extractor.getPrimaryInputs();
00697     spdlog::info("Found {} Primary Inputs:", inputs.size());
00698     for (const auto &name : inputs) {
00699       spdlog::info("  - {}", name);
00700     }
00701
00702     const auto &outputs = extractor.getPrimaryOutputs();
00703     spdlog::info("Found {} Primary Outputs:", outputs.size());
00704     for (const auto &name : outputs) {
00705       spdlog::info("  - {}", name);
00706     }
00707
00708     const auto &wires = extractor.getInternalWires();
00709     spdlog::info("Found {} Internal Wires:", wires.size());
00710     for (const auto &name : wires) {
00711       spdlog::info("  - {}", name);
00712     }
00713
00714     const auto &gates = extractor.getExtractedGates();
00715     spdlog::info("Found {} Gate Drivers:", gates.size());
00716     for (const auto &pair : gates) {
00717       const std::string &outputNet = pair.first;
```

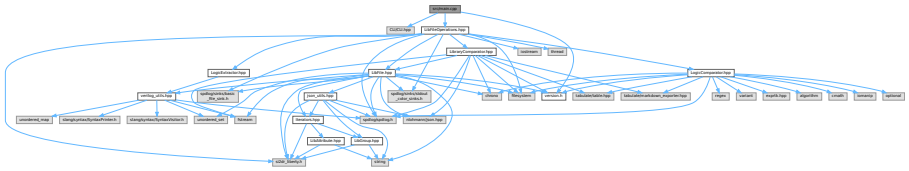
```
00718       const GateInfo &info = pair.second;
00719       std::string inputsStr = "";
00720       for (size_t i = 0; i < info.inputSignals.size(); ++i) {
00721         inputsStr += info.inputSignals[i] + (i == info.inputSignals.size() - 1 ? "" : ", ");
00722       }
00723       spdlog::info("  - Output: {} <= Driven by: {} ({}) Inputs: [{}]", outputNet,
00724                    info.gateTypeName, slang::parsing::toString(info.kind), inputsStr);
00725     }
00726     spdlog::info("--- End Netlist Info Extraction ---");
00727
00728   } catch (const std::exception &e) {
00729     spdlog::error("Exception during Verilog parsing or info extraction: {}", e.what());
00730   } catch (...) {
00731     spdlog::error("Unknown exception during Verilog parsing or info extraction.");
00732   }
00733 }
00734
00735 // --- (Step 2: Extract Logic Expressions) ---
00736
00755 std::map<std::string, std::string> extractLogicFromVerilog(const std::string &verilog_file,
00756                                                            const std::string &cell) {
00757   spdlog::info("--- Starting Logic Expression Extraction for cell: '{}' ---", cell);
00758   std::map<std::string, std::string> logicMap; // Default empty map
00759
00760   try {
00761     auto result = slang::syntax::SyntaxTree::fromFile(verilog_file);
00762     if (!result) {
00763       spdlog::error("Error parsing Verilog file '{}'. Cannot extract logic.", verilog_file);
00764       return logicMap;
00765     }
00766
00767     spdlog::info("Successfully parsed Verilog file.");
00768     std::shared_ptr<slang::syntax::SyntaxTree> tree = result.value();
00769
00770     LogicExtractor extractor(cell);
00771     tree->root().visit(extractor); // Populate extractor's internal state
00772
00773     // Now, call the method to derive and get the expressions
00774     logicMap = extractor.getLogicExpressions();
00775
00776   } catch (const std::exception &e) {
00777     spdlog::error("Exception during Verilog parsing or logic extraction: {}", e.what());
00778   } catch (...) {
00779     spdlog::error("Unknown exception during Verilog parsing or logic extraction.");
00780   }
00781
00782   if (logicMap.empty()) {
00783     spdlog::warn("Logic extraction finished, but no expressions were derived for cell '{}'. Check "
00784                  "if cell exists and is correctly defined.",
00785                  cell);
00786   } else {
00787     spdlog::info("Logic extraction completed for cell '{}'. Found {} output expression strings",
00788                  cell, logicMap.size());
00789   }
00790
00791   spdlog::info("--- End Logic Expression Extraction ---");
00792   return logicMap;
00793 }
```

7.43 src/main.cpp File Reference

This file contains the main function for the ZlibValidation tool.

```
#include "CLI/CLI.hpp"
#include "LibFileOperations.hpp"
#include "version.h"
```

Include dependency graph for main.cpp:



Functions

- int main (int argc, char *argv[])

7.43.1 Detailed Description

This file contains the main function for the ZlibValidation tool.

The ZlibValidation tool is a command-line application that provides several functionalities for processing and validating Liberty files, including parsing, monotonicity checking, comparison, supercell generation, Verilog/SPICE netlist generation, functional equivalence check, and a utility to clear generated files. It uses the CLI11 library for command-line argument parsing and spdlog for logging.

The main function parses command-line arguments using CLI11 to determine the desired operation and its parameters. It then calls the appropriate functions to perform the requested task. The tool supports processing multiple Liberty files in parallel using multi-threading for monotonicity checking, supercell generation, Verilog generation, and SPICE generation.

Parameters

argc	The number of command-line arguments.
argv	An array of command-line argument strings.

Returns

0 if the program executes successfully, or an error code otherwise.

The main function performs the following steps:

1. Initializes command-line argument parsing using CLI11.

2. Defines subcommands for each supported operation:
    - `parse`: Parses a Liberty file and writes JSON to a file.
    - `mono`: Checks the monotonicity of timing arc values in a Liberty file.
    - `compare`: Compares two Liberty files and reports differences.
    - `supercell`: Generates supercells for a given Liberty file.
    - `zlibboost`: Runs the ZlibBoost tool for multi-threaded library processing.
    - `clear`: Clears log, JSON, map, markdown, Verilog, and SPICE files in the current directory.
    - `verilog`: Generates Verilog netlist for a given Liberty file.
    - `spice`: Generates SPICE netlist for a given Liberty file.
    - `func`: Checks functional equivalence of two Liberty or Verilog files.
3. Parses the command-line arguments using CLI11.
4. Calls the appropriate function based on the selected subcommand.
5. Logs the program's exit status and timestamp.

Note

The tool relies on external libraries such as CLI11, spdlog, and potentially others depending on the specific operation being performed. Ensure that these libraries are properly installed and configured before running the tool.

Definition in file main.cpp.

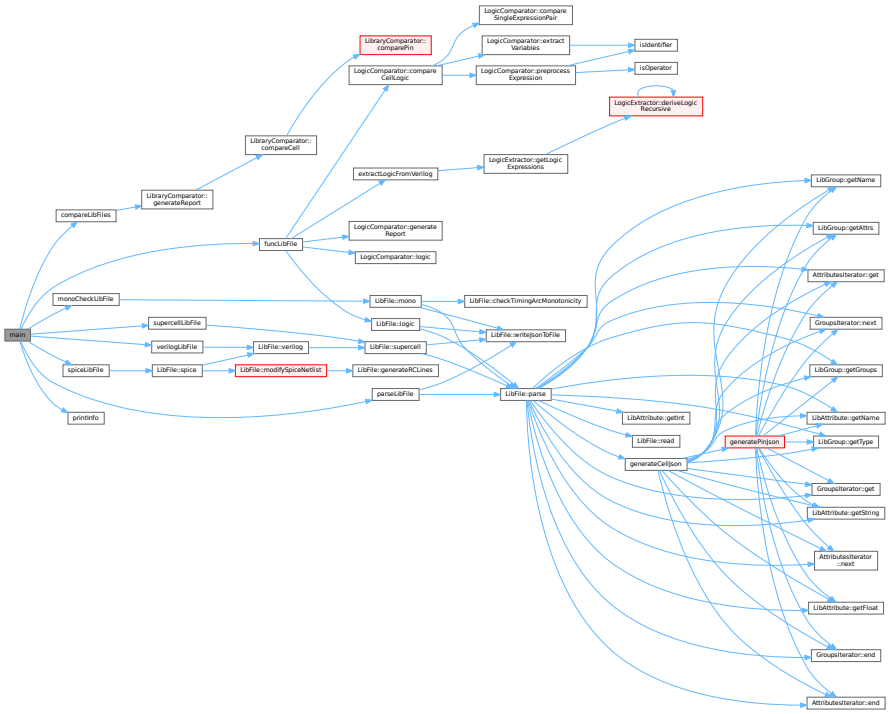
7.43.2 Function Documentation

7.43.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Definition at line 49 of file main.cpp.

Here is the call graph for this function:



7.44 main.cpp

Go to the documentation of this file.

```
00001 // ./src/main.cpp
00002
00003 #include "CLI/CLI.hpp"
00004
00005 #include "LibFileOperations.hpp"
00006 #include "version.h"
00007
00049 int main(int argc, char *argv[]) {
00050   // Parse command line arguments
00051   std::vector<std::string> library_paths; // Support multiple files
00052   std::string log_file_name = "";
00053
00054   // Command line parameter parsing using CLI11
00055   CLI::App app{APP_NAME};
00056   app.set_version_flag("-v,--version", APP_VERSION);
00057
00058   // Add subcommand for parse mode
00059   CLI::App *parse_cmd =
00060       app.add_subcommand("parse", "Parse the Liberty file and write JSON to a file");
00061   parse_cmd->add_option("library_path", library_paths, "Specify the library file to process")
00062       ->check(CLI::ExistingFile)
00063       ->required();
00064   parse_cmd->add_option("-l,--log", log_file_name,
00065                         "Specify the log file name. Default: <basename>.parse.log");
00066   parse_cmd->callback([&] {
00067     printInfo();
00068     // Check if multi files
00069     if (library_paths.size() > 1) {
00070       spdlog::info("Running sequential parsing for {} files.", library_paths.size());
```

```
00071       spdlog::info("Each library will write to its own log file.");
00072       // Sequential parsing
00073       for (const auto &library_path : library_paths) {
00074         parseLibFile(library_path, log_file_name = "");
00075       }
00076     } else {
00077       parseLibFile(library_paths[0], log_file_name);
00078     }
00079   });
00080
00081   // Add subcommand for mono check mode
00082   bool is_slew = false;
00083   CLI::App *mono_cmd = app.add_subcommand("mono", "Check the monotonicity of timing arc values");
00084   mono_cmd->add_option("library_path", library_paths, "Specify the library file to process")
00085       ->check(CLI::ExistingFile)
00086       ->required();
00087   mono_cmd->add_option("-l,--log", log_file_name,
00088                        "Specify the log file name. Default: <basename>.mono.log");
00089   mono_cmd->add_flag("-s,--slew", is_slew,
00090                      "Specify that monotonicity checks also include input slew.");
00091   mono_cmd->callback([&] {
00092     printInfo();
00093     // Check if multi files
00094     if (library_paths.size() > 1) {
00095       spdlog::info("Running multi-threaded monotonicity check for {} files.", library_paths.size());
00096       spdlog::info("Each thread will write to its own log file.");
00097
00098       // Sequential json file check
00099       for (const auto &library_path : library_paths) {
00100         if (!std::filesystem::exists(std::filesystem::path(library_path).stem().string() +
00101                                      ".json")) {
00102           spdlog::info("JSON file not found for '{}'. Parsing Liberty file first.", library_path);
00103           parseLibFile(library_path, log_file_name = "");
00104         }
00105       }
00106       spdlog::info("All JSON files prepared.");
00107       // Parallel monotonicity check
00108       std::vector<std::thread> threads;
00109       for (const auto &library_path : library_paths) {
00110         threads.emplace_back(monoCheckLibFile, library_path, log_file_name = "", is_slew);
00111       }
00112       for (auto &thread : threads) {
00113         thread.join();
00114       }
00115       spdlog::info("All threads completed.");
00116     } else {
00117       monoCheckLibFile(library_paths[0], log_file_name, is_slew);
00118     }
00119   });
00120
00121   // Add subcommand for compare mode
00122   std::string ref_lib, comp_lib, report_file_name;
00123   CLI::App *compare_cmd = app.add_subcommand(
00124       "compare", "Compare the comparison library against the reference one and report differences");
00125   compare_cmd->add_option("--ref", ref_lib, "Specify the reference library file")
00126       ->check(CLI::ExistingFile)
00127       ->required();
00128   compare_cmd->add_option("--comp", comp_lib, "Specify the comparison library file")
00129       ->check(CLI::ExistingFile)
00130       ->required();
00131   double abstol = 0.002;
00132   compare_cmd->add_option("--abstol", abstol,
00133                           "Specify the absolute tolerance for comparison. Default: 0.002ns");
00134   double reltol = 0.02;
00135   compare_cmd->add_option("--reltol", reltol,
00136                           "Specify the relative tolerance for comparison. Default: 0.02/2.0%");
00137   compare_cmd->add_option("--report", report_file_name,
00138                           "Specify the report file name. Default: <comp_lib>.cmp.md");
00139   compare_cmd->callback([&] {
```

```
00140     printInfo();
00141     compareLibFiles(ref_lib, comp_lib, reltol, abstol, report_file_name);
00142   });
00143 
00144   // Add subcommand for supercell generation
00145   int chain_length = 1;
00146   CLI::App *supercell_cmd =
00147       app.add_subcommand("supercell", "Generate supercells for the given Liberty file");
00148   supercell_cmd->add_option("library_path", library_paths, "Specify the library file to process")
00149       ->check(CLI::ExistingFile)
00150       ->required();
00151   supercell_cmd->add_option("-l,--log", log_file_name,
00152                             "Specify the log file name. Default: <basename>.supercell.log");
00153   supercell_cmd->add_option("-c,--chain", chain_length,
00154                             "Specify the chain length for supercell generation. Default: 1");
00155   std::vector<std::string> cell_names = {}; // "CMPE42D1" "AN2D0", "DFQD1"
00156   supercell_cmd->add_option("--cells", cell_names,
00157                             "Specify the cell names to generate supercells for");
00158   supercell_cmd->callback([&] {
00159     printInfo();
00160     // Check if multi files
00161     if (library_paths.size() > 1) {
00162       spdlog::info("Running multi-threaded supercell generation for {} files.",
00163                    library_paths.size());
00164       spdlog::info("Each library will write to its own log file.");
00165 
00166       // Sequential json file check
00167       for (const auto &library_path : library_paths) {
00168         if (!std::filesystem::exists(std::filesystem::path(library_path).stem().string() +
00169                                      ".json")) {
00170           spdlog::info("JSON file not found for '{}'. Parsing Liberty file first.", library_path);
00171           parseLibFile(library_path, log_file_name = "");
00172         }
00173       }
00174       spdlog::info("All JSON files prepared.");
00175       // Parallel supercell generation
00176       std::vector<std::thread> threads;
00177       for (const auto &library_path : library_paths) {
00178         threads.emplace_back(supercellLibFile, library_path, log_file_name = "", chain_length,
00179                              cell_names);
00180       }
00181       for (auto &thread : threads) {
00182         thread.join();
00183       }
00184       spdlog::info("All threads completed.");
00185     } else {
00186       supercellLibFile(library_paths[0], log_file_name, chain_length, cell_names);
00187     }
00188   });
00189 
00190   // Add subcommand for zlibboost
00191   CLI::App *zlibboost_cmd =
00192       app.add_subcommand("zlibboost", "ZlibBoost - Multi-threaded Library Processing Tool");
00193   std::string config_dir = "/home/songzx/Projects/zlibboost/config.tcl";
00194   std::string python_dir = "/home/guocj/anaconda3/envs/myenv/bin/python";
00195   std::string main_py_dir = "/home/songzx/Projects/zlibboost/zlibboost.py";
00196   zlibboost_cmd->add_option(
00197       "-c, --config", config_dir,
00198       "Specify the configuration TCL file. Default: /home/songzx/Projects/zlibboost/config.tcl");
00199   zlibboost_cmd->add_option(
00200       "--python", python_dir,
00201       "Specify the python directory. Default: /home/guocj/anaconda3/envs/myenv/bin/python");
00202   zlibboost_cmd->add_option("--main", main_py_dir,
00203                             "Specify the main python script directory. Default: "
00204                             "/home/songzx/Projects/zlibboost/zlibboost.py");
00205   zlibboost_cmd->callback([&] {
00206     printInfo();
00207     // Run the ZlibBoost tool
00208     std::string command = python_dir + " " + main_py_dir + " -c " + config_dir;
```

```
00209     spdlog::info("Running ZlibBoost with command: '{}'", command);
00210     int ret = std::system(command.c_str());
00211     if (ret == 0) {
00212       spdlog::info("ZlibBoost completed successfully.");
00213     } else {
00214       spdlog::error("ZlibBoost failed with return code: {}", ret);
00215     }
00216   });
00217
00218   // Add subcommand for clear
00219   CLI::App *clear_cmd = app.add_subcommand(
00220       "clear", "Clear the log, JSON, map, markdown, Verilog, SPICE files in this directory");
00221   clear_cmd->callback([&] {
00222     printInfo();
00223     std::filesystem::path current_dir = std::filesystem::current_path();
00224     for (const auto &entry : std::filesystem::directory_iterator(current_dir)) {
00225       if (entry.path().extension() == ".log" || entry.path().extension() == ".json" ||
00226           entry.path().extension() == ".map" || entry.path().extension() == ".md" ||
00227           entry.path().extension() == ".v" || entry.path().extension() == ".spi") {
00228         spdlog::info("Removing file: '{}'", entry.path().string());
00229         std::filesystem::remove(entry.path());
00230       }
00231     }
00232     spdlog::info("All log, JSON, map, markdown files cleared.");
00233   });
00234
00235   // Add subcommand for Verilog generation
00236   CLI::App *verilog_cmd =
00237       app.add_subcommand("verilog", "Generate Verilog file for given Liberty file");
00238   verilog_cmd->add_option("library_path", library_paths, "Specify the library file to process")
00239       ->check(CLI::ExistingFile)
00240       ->required();
00241   verilog_cmd->add_option("-l,--log", log_file_name,
00242                           "Specify the log file name. Default: <basename>.verilog.log");
00243   verilog_cmd->add_option("-c,--chain", chain_length,
00244                           "Specify the chain length for verilog generation. Default: 1");
00245   verilog_cmd->add_option("--cells", cell_names, "Specify the cell names to generate Verilog for");
00246   verilog_cmd->callback([&] {
00247     printInfo();
00248     // Check if multi files
00249     if (library_paths.size() > 1) {
00250       spdlog::info("Running multi-threaded Verilog generation for {} files.", library_paths.size());
00251       spdlog::info("Each library will write to its own log file.");
00252
00253       // Sequential json file check
00254       for (const auto &library_path : library_paths) {
00255         if (!std::filesystem::exists(std::filesystem::path(library_path).stem().string() +
00256                                      ".json")) {
00257           spdlog::info("JSON file not found for '{}'. Parsing Liberty file first.", library_path);
00258           parseLibFile(library_path, log_file_name = "");
00259         }
00260       }
00261       spdlog::info("All JSON files prepared.");
00262       // Parallel Verilog generation
00263       std::vector<std::thread> threads;
00264       for (const auto &library_path : library_paths) {
00265         threads.emplace_back(verilogLibFile, library_path, log_file_name = "", chain_length,
00266                              cell_names);
00267       }
00268       for (auto &thread : threads) {
00269         thread.join();
00270       }
00271       spdlog::info("All threads completed.");
00272     } else {
00273       verilogLibFile(library_paths[0], log_file_name, chain_length, cell_names);
00274     }
00275   });
00276
00277   // Add subcommand for SPICE generation
```

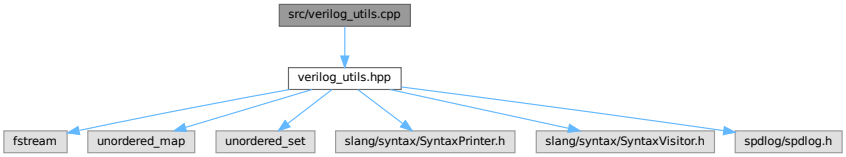
```
00278   CLI::App *spice_cmd = app.add_subcommand("spice", "Generate SPICE file for given Liberty file");
00279   spice_cmd->add_option("library_path", library_paths, "Specify the library file to process")
00280       ->check(CLI::ExistingFile)
00281       ->required();
00282   spice_cmd->add_option("-l,--log", log_file_name,
00283                         "Specify the log file name. Default: <basename>.spice.log");
00284   spice_cmd->add_option("-c,--chain", chain_length,
00285                         "Specify the chain length for SPICE generation. Default: 1");
00286   spice_cmd->add_option("--cells", cell_names, "Specify the cell names to generate SPICE for");
00287   std::string verilog_lib_file =
00288       "/home/songzx/examples/mypdk/TSMC65/TSMC65NM_CLN65LP_STDIO_STDCELL/tcbn65lp_220a/"
00289       "0396011_20170308/TSMCHOME/digital/Front_End/verilog/tcbn65lp.v";
00290   std::string spice_lib_file =
00291       "/home/songzx/examples/mypdk/TSMC65/TSMC65NM_CLN65LP_STDIO_STDCELL/tcbn65lp_220a/"
00292       "0396011_20170308/TSMCHOME/digital/Back_End/lpe_spice/tcbn65lp_200a/tcbn65lp_200a_lpe.spi";
00293   spice_cmd->add_option("--vl", verilog_lib_file,
00294                         "Specify the location of the Verilog primitive library file");
00295   spice_cmd->add_option(
00296       "--sl", spice_lib_file,
00297       "Specify the location of the SPICE library file to be included in the output");
00298   spice_cmd->callback([&] {
00299     printInfo();
00300     // Check if multi files
00301     if (library_paths.size() > 1) {
00302       spdlog::info("Running multi-threaded SPICE generation for {} files.", library_paths.size());
00303       spdlog::info("Each library will write to its own log file.");
00304
00305       // Sequential json file check
00306       for (const auto &library_path : library_paths) {
00307         if (!std::filesystem::exists(std::filesystem::path(library_path).stem().string() +
00308                                      ".json")) {
00309           spdlog::info("JSON file not found for '{}'. Parsing Liberty file first.", library_path);
00310           parseLibFile(library_path, log_file_name = "");
00311         }
00312       }
00313       spdlog::info("All JSON files prepared.");
00314       // Parallel SPICE generation
00315       std::vector<std::thread> threads;
00316       for (const auto &library_path : library_paths) {
00317         threads.emplace_back(spiceLibFile, library_path, log_file_name = "", chain_length,
00318                              cell_names, verilog_lib_file, spice_lib_file);
00319       }
00320       for (auto &thread : threads) {
00321         thread.join();
00322       }
00323       spdlog::info("All threads completed.");
00324     } else {
00325       spiceLibFile(library_paths[0], log_file_name, chain_length, cell_names, verilog_lib_file,
00326                    spice_lib_file);
00327     }
00328   });
00329
00330   // Add subcommand for funtional equivalence check
00331   CLI::App *func_cmd = app.add_subcommand(
00332       "func", "Check functional equivalence of two Liberty files or Verilog files");
00333   std::string ref_file, comp_file;
00334   func_cmd->add_option("--ref", ref_file, "Specify the reference Liberty or Verilog file")
00335       ->check(CLI::ExistingFile)
00336       ->required();
00337   func_cmd->add_option("--comp", comp_file, "Specify the comparison Liberty or Verilog file")
00338       ->check(CLI::ExistingFile)
00339       ->required();
00340   func_cmd->add_option("--cells", cell_names,
00341                        "Specify the cell names to check functional equivalence for");
00342   func_cmd->add_option("--report", report_file_name,
00343                        "Specify the report file name. Default: <comp_lib>.cmp.md");
00344
00345   func_cmd->callback([&] {
00346     printInfo();
```

```
00347     funcLibFile(ref_file, comp_file, cell_names, report_file_name);
00348   });
00349
00350   CLI11_PARSE(app, argc, argv);
00351
00352   // End of program
00353   char hostname[256];
00354   gethostname(hostname, sizeof(hostname));
00355
00356   auto now = std::chrono::system_clock::now();
00357   auto time_now = std::chrono::system_clock::to_time_t(now);
00358   std::stringstream ss;
00359   ss << std::put_time(std::localtime(&time_now), "%c");
00360
00361   spdlog::info("ZlibValidation exited on '{}' at {}", hostname, ss.str());
00362   return 0;
00363 }
```

7.45 src/verilog_utils.cpp File Reference

`#include "verilog_utils.hpp"`

Include dependency graph for verilog_utils.cpp:



Functions

- void getAST (const std::string &verilog_file, const std::string &cell)

7.45.1 Function Documentation

7.45.1.1 getAST()

```
void getAST (
                const std::string & verilog_file,
                const std::string & cell )
```

Definition at line 544 of file verilog_utils.cpp.

Here is the call graph for this function:

7.46 verilog_utils.cpp

Go to the documentation of this file.

```
00001 // verilog_utils.cpp
00002
00003 #include "verilog_utils.hpp"
00004
00005 // Adding a generic handler method to record all visited nodes and increment the depth counter
00006 void VerilogVisitor::handle(const slang::syntax::SyntaxNode &node) {
00007   std::string indent(depth_ * 2, ' ');
00008   spdlog::debug("{}Node: {}", indent, slang::syntax::toString(node.kind));
00009
00010   // Increase depth, visit child nodes, then decrease depth
00011   depth_++;
00012   visitDefault(node);
00013   depth_--;
00014 }
00015
00016 // Handle module declarations
00017 void VerilogVisitor::handle(const slang::syntax::ModuleDeclarationSyntax &module) {
00018   std::string indent(depth_ * 2, ' ');
00019
00020   if (module.header && module.header->name.valid()) {
00021     std::string_view moduleName = module.header->name.valueText();
00022     spdlog::debug("{}Module Name:{}", indent, moduleName);
00023
00024     inTargetModule_ = !targetCell_.empty() && moduleName == targetCell_;
00025     if (!inTargetModule_) {
00026       return;
00027     }
00028
00029     // If a specific module name is specified, check if it matches
00030     if (inTargetModule_) {
00031       spdlog::info("{}Found target module: {}", indent, targetCell_);
00032       // Print the module ports
00033       if (module.header->ports) {
00034         spdlog::info("{}Ports: {}", indent, module.header->ports->toString());
00035       }
00036     }
00037   }
00038
00039   // Continue processing child nodes
00040   depth_++;
00041   visitDefault(module);
00042   depth_--;
00043 }
00044
00045 // Handle port declarations
00046 void VerilogVisitor::handle(const slang::syntax::PortDeclarationSyntax &portDecl) {
00047   if (!inTargetModule_) {
```

```
00048     return;
00049   }
00050
00051   std::string indent = std::string(depth_ * 2, ' ');
00052
00053   // Get port direction
00054   std::string direction = "unknown";
00055   if (portDecl.header) {
00056     // Try to convert the header to different types of port headers
00057     if (auto *varPort = portDecl.header->as_if<slang::syntax::VariablePortHeaderSyntax>()) {
00058       if (varPort->direction) {
00059         direction = varPort->direction.valueText();
00060       }
00061     } else if (auto *netPort = portDecl.header->as_if<slang::syntax::NetPortHeaderSyntax>()) {
00062       if (netPort->direction) {
00063         direction = netPort->direction.valueText();
00064       }
00065     }
00066
00067     // Get port name
00068     for (const auto &declarator : portDecl.declarators) {
00069       if (declarator->name.valid()) {
00070         std::string_view portName = declarator->name.valueText();
00071         spdlog::info("{}Port Name: {} ({})", indent, portName, direction);
00072       }
00073     }
00074   }
00075 }
00076
00077 // Handle hierarchical instantiation (module instance)
00078 void VerilogVisitor::handle(const slang::syntax::HierarchyInstantiationSyntax &hierarchyInst) {
00079   if (!inTargetModule_) {
00080     return;
00081   }
00082
00083   std::string indent(depth_ * 2, ' ');
00084
00085   if (hierarchyInst.type.valid()) {
00086     std::string_view instType = hierarchyInst.type.valueText();
00087     spdlog::info("{}Hierarchy Instance Type: {}", indent, instType);
00088
00089     // Safely print parameter values (if any)
00090     try {
00091       if (hierarchyInst.parameters) {
00092         spdlog::info("{}Parameters:", indent);
00093         for (const auto &param : hierarchyInst.parameters->parameters) {
00094           if (!param)
00095             continue; // Null pointer check
00096
00097           if (auto *orderedParam = param->as_if<slang::syntax::OrderedParamAssignmentSyntax>()) {
00098             if (orderedParam->expr) {
00099               spdlog::info("{}  Parameter: {}", indent, orderedParam->expr->toString());
00100             }
00101           } else if (auto *namedParam = param->as_if<slang::syntax::NamedParamAssignmentSyntax>()) {
00102             if (namedParam->name.valid() && namedParam->expr) {
00103               spdlog::info("{}  Parameter {}: {}", indent, namedParam->name.valueText(),
00104                            namedParam->expr->toString());
00105             }
00106           }
00107         }
00108       }
00109     } catch (const std::exception &e) {
00110       spdlog::warn("{}Exception while processing parameters: {}", indent, e.what());
00111     }
00112
00113     // Safely handle instances
00114     try {
00115       for (const auto &instance : hierarchyInst.instances) {
00116         if (!instance) {
```

```
00117           spdlog::warn("{}Invalid instance", indent);
00118           continue;
00119         } else if (!instance->decl) {
00120           spdlog::warn("{}Invalid instance declaration", indent);
00121           // continue;
00122         } else {
00123           try {
00124             if (instance->decl->name.valid()) {
00125               std::string_view instName = instance->decl->name.valueText();
00126               spdlog::info("{}Instance Name: {}", indent, instName);
00127             }
00128           } catch (...) {
00129             spdlog::debug("{}Error printing name", indent);
00130           }
00131
00132           // Safely print dimensions
00133           try {
00134             if (!instance->decl->dimensions.empty()) {
00135               spdlog::info("{}  Dimensions: {}", indent, instance->decl->dimensions.toString());
00136             }
00137           } catch (...) {
00138             spdlog::debug("{}Error printing dimensions", indent);
00139           }
00140         }
00141
00142         // Safely print port connections
00143         spdlog::info("{}  Port connections:", indent);
00144         for (const auto &conn : instance->connections) {
00145           if (!conn) {
00146             spdlog::warn("{}    Null connection", indent);
00147             continue; // Null pointer check
00148           }
00149
00150           try {
00151             // Use as_if method for safe type conversion
00152             if (auto *ordered = conn->as_if<slang::syntax::OrderedPortConnectionSyntax>()) {
00153               if (ordered->expr) {
00154                 spdlog::info("{}    Ordered connection: {}", indent, ordered->expr->toString());
00155               } else {
00156                 spdlog::info("{}    Ordered connection: <empty>", indent);
00157               }
00158             } else if (auto *named = conn->as_if<slang::syntax::NamedPortConnectionSyntax>()) {
00159               if (named->name.valid()) {
00160                 std::string exprStr = named->expr ? named->expr->toString() : "<empty>";
00161                 spdlog::info("{}    .{}({})", indent, named->name.valueText(), exprStr);
00162               }
00163             } else if (conn->as_if<slang::syntax::EmptyPortConnectionSyntax>()) {
00164               spdlog::info("{}    <empty connection>", indent);
00165             } else if (conn->as_if<slang::syntax::WildcardPortConnectionSyntax>()) {
00166               spdlog::info("{}    .* (wildcard connection)", indent);
00167             } else {
00168               spdlog::info("{}    Unknown connection type: {}", indent,
00169                            slang::syntax::toString(conn->kind));
00170             }
00171           } catch (const std::exception &e) {
00172             spdlog::warn("{}Exception processing connection: {}", indent, e.what());
00173           } catch (...) {
00174             spdlog::warn("{}Unknown exception processing connection", indent);
00175           }
00176         }
00177       }
00178     } catch (const std::exception &e) {
00179       spdlog::warn("{}Exception while processing instances: {}", indent, e.what());
00180     }
00181
00182     // Continue processing child nodes
00183     depth_++;
00184     visitDefault(hierarchyInst);
00185     depth_--;
```

```
00186   }
00187 }
00188 /*
00189 // Handle primitive gate instantiation
00190 void VerilogVisitor::handle(const slang::syntax::PrimitiveInstantiationSyntax &primitiveInst) {
00191 if (!inTargetModule_) {
00192 return;
00193 }
00194
00195 std::string indent(depth_ * 2, ' ');
00196
00197 // Get primitive gate type
00198 if (primitiveInst.type.valid()) {
00199 std::string_view gateType = primitiveInst.type.valueText();
00200 spdlog::info("{}Primitive Gate: {}", indent, gateType);
00201
00202 // Print delay information (if any)
00203 if (primitiveInst.delay) {
00204 spdlog::info("{}Delay: {}", indent, primitiveInst.delay->toString());
00205 }
00206
00207 // Print strength information (if any)
00208 if (primitiveInst.strength) {
00209 spdlog::info("{}Strength: {}", indent, primitiveInst.strength->toString());
00210 }
00211
00212 // Print each instance
00213 for (const auto &instance : primitiveInst.instances) {
00214 // Primitive gates usually don't have names, but if they do, print them
00215 if (instance->decl && instance->decl->name.valid()) {
00216 std::string_view instName = instance->decl->name.valueText();
00217 spdlog::info("{}  Gate instance: {}", indent, instName);
00218 }
00219
00220 // Print connections
00221 spdlog::info("{}  Gate connections:", indent);
00222 for (size_t i = 0; i < instance->connections.size(); ++i) {
00223 const auto &conn = instance->connections[i];
00224 // For gate-level instantiation, connections are usually ordered
00225 if (auto *ordered = conn->as_if<slang::syntax::OrderedPortConnectionSyntax>()) {
00226 if (ordered->expr) {
00227 // The first is usually the output, the rest are inputs
00228 std::string portType = (i == 0) ? "output" : "input";
00229 spdlog::info("{}    {} {}: {}", indent, portType, i, ordered->expr->toString());
00230 }
00231 }
00232 }
00233 }
00234
00235 // Continue to traverse deeper when specific standard gate type
00236 // Create a set of safe standard gate types
00237 static const std::unordered_set<std::string_view> safeGateTypes = {
00238 "and",   "or",      "nand",    "nor",      "xor",     "xnor",   "not",
00239 "buf",   "bufif0",  "bufif1",  "notif0",   "notif1",  "pullup", "pulldown",
00240 "cmos",  "rcmos",   "nmos",    "pmos",     "rnmos",   "rpmos",  "tran",
00241 "rtran", "tranif0", "tranif1", "rtranif0", "rtranif1"};
00242
00243 if (safeGateTypes.find(gateType) != safeGateTypes.end()) {
00244 // Continue processing child nodes
00245 depth_++;
00246 visitDefault(primitiveInst);
00247 depth_--;
00248 } else {
00249 spdlog::warn("{}Skipping deeper traversal of primitive: {}", indent, gateType);
00250 }
00251
00252 } else {
00253 spdlog::warn("{}Primitive instantiation missing gate type", indent);
00254 }
```

```
00255 }
00256 */
00257 // Handle specify block
00258 void VerilogVisitor::handle(const slang::syntax::SpecifyBlockSyntax &specifyBlock) {
00259   if (!inTargetModule_) {
00260     return;
00261   }
00262
00263   std::string indent(depth_ * 2, ' ');
00264   spdlog::info("{}Specify Block:", indent);
00265
00266   // Iterate through all path declarations
00267   for (const auto &item : specifyBlock.items) {
00268     if (auto *pathDecl = item->as_if<slang::syntax::PathDeclarationSyntax>()) {
00269       if (pathDecl->desc) {
00270         std::string pathSrc;
00271         std::string pathDst;
00272
00273         // Get path source
00274         if (!pathDecl->desc->inputs.empty()) {
00275           // Get the first input
00276           if (auto *identifier =
00277                   pathDecl->desc->inputs[0]->as_if<slang::syntax::IdentifierNameSyntax>()) {
00278             if (identifier->identifier.valid()) {
00279               pathSrc = identifier->identifier.valueText();
00280             }
00281           }
00282         }
00283
00284         // Get path destination
00285         if (pathDecl->desc->suffix) {
00286           if (auto *simpleSuffix =
00287                   pathDecl->desc->suffix->as_if<slang::syntax::SimplePathSuffixSyntax>()) {
00288             if (!simpleSuffix->outputs.empty()) {
00289               if (auto *identifier =
00290                       simpleSuffix->outputs[0]->as_if<slang::syntax::IdentifierNameSyntax>()) {
00291                 if (identifier->identifier.valid()) {
00292                   pathDst = identifier->identifier.valueText();
00293                 }
00294               }
00295             } else if (auto *edgeSuffix =
00296                            pathDecl->desc->suffix
00297                                ->as_if<slang::syntax::EdgeSensitivePathSuffixSyntax>()) {
00298               if (!edgeSuffix->outputs.empty()) {
00299                 if (auto *identifier =
00300                         edgeSuffix->outputs[0]->as_if<slang::syntax::IdentifierNameSyntax>()) {
00301                   if (identifier->identifier.valid()) {
00302                     pathDst = identifier->identifier.valueText();
00303                   }
00304                 }
00305               }
00306             }
00307           }
00308
00309           // Construct path string
00310           std::string pathStr = pathSrc + " => " + pathDst;
00311
00312           // Get path delay
00313           std::string delayStr = "(";
00314           for (size_t i = 0; i < pathDecl->delays.size(); ++i) {
00315             if (i > 0)
00316               delayStr += ", ";
00317             delayStr += pathDecl->delays[i]->toString();
00318           }
00319           delayStr += ")";
00320
00321           spdlog::info("{}  Path: {} = {}", indent, pathStr, delayStr);
00322         }
00323       }
```

```
00324       // Can add handling for ConditionalPathDeclarationSyntax and IfNonePathDeclarationSyntax
00325       else if (auto *condPath = item->as_if<slang::syntax::ConditionalPathDeclarationSyntax>()) {
00326         if (condPath->path && condPath->predicate) {
00327           spdlog::info("{}  Conditional Path (if {})", indent, condPath->predicate->toString());
00328           // Recursively handle internal path
00329           handle(*condPath->path);
00330         }
00331       } else if (auto *ifNonePath = item->as_if<slang::syntax::IfNonePathDeclarationSyntax>()) {
00332         if (ifNonePath->path) {
00333           spdlog::info("{}  If-None Path", indent);
00334           // Recursively handle internal path
00335           handle(*ifNonePath->path);
00336         }
00337       }
00338     }
00339
00340     // Continue processing child nodes
00341     depth_++;
00342     visitDefault(specifyBlock);
00343     depth_--;
00344   }
00345 }
00346
00347 // Handle module declarations, only keep the target module
00348 void CellExtractor::handle(const slang::syntax::ModuleDeclarationSyntax &module) {
00349   if (module.header && module.header->name.valid()) {
00350     std::string_view moduleName = module.header->name.valueText();
00351
00352     // If it is the target module, mark as found, otherwise remove it
00353     if (!targetCell_.empty() && moduleName == targetCell_) {
00354       foundTarget_ = true;
00355       // Do not modify, keep this module
00356     } else {
00357       // Remove non-target modules
00358       remove(module);
00359     }
00360   }
00361 }
00362
00363 // Get result
00364 bool CellExtractor::foundTargetCell()const { return foundTarget_; }
00365
00366 // Handle module declarations
00367 void CellPrinter::handle(const slang::syntax::ModuleDeclarationSyntax &module) {
00368   if (module.header && module.header->name.valid()) {
00369     std::string_view moduleName = module.header->name.valueText();
00370
00371     // Check if it is the target module
00372     if (!targetCell_.empty() && moduleName == targetCell_) {
00373       foundTarget_ = true;
00374
00375       // Print module definition
00376       out_ << "`timescale 1ns/10ps\n";
00377       out_ << module.toString();
00378
00379       return; // Do not traverse further
00380     }
00381   }
00382
00383   // Continue traversing other modules
00384   visitDefault(module);
00385 }
00386
00396 void ModuleRewriter::handle(const slang::syntax::SyntaxNode &node) {
00397   std::string indent(depth_ * 2, ' ');
00398   logger_->debug("{}Node: {}", indent, slang::syntax::toString(node.kind));
00399
00400   // Continue processing child nodes
00401   depth_++;
```

```
00402   visitDefault(node);
00403   depth_--;
00404 }
00405
00406 void ModuleRewriter::handle(const slang::syntax::ModuleDeclarationSyntax &module) {
00407   logger_->debug("Processing module: {}", module.header->name.valueText());
00408
00409   // Create intermediate wires for connections between instances
00410   for (int i = 0; i < instance_count_ - 1; i++) {
00411     auto &newNetNode = parse("\n  wire OP_" + std::to_string(i) + ";");
00412     insertAtBack(module.members, newNetNode);
00413     logger_->debug("Added intermediate wire: {}", newNetNode.toString());
00414   }
00415
00416   // Get critical input port & output port from
00417   // the module name pattern: CELLNAME__X#__CRITICALPORT__OUTPUTPORT
00418   std::string criticalInputPort = "";
00419   std::string criticalOutputPort = "";
00420   if (!this->moduleName_.empty()) {
00421     size_t firstSep = this->moduleName_.find("__");
00422     if (firstSep != std::string::npos) {
00423       size_t secondSep = this->moduleName_.find("__", firstSep + 2);
00424       if (secondSep != std::string::npos) {
00425         size_t thirdSep = this->moduleName_.find("__", secondSep + 2);
00426         if (thirdSep != std::string::npos) {
00427           criticalInputPort = this->moduleName_.substr(secondSep + 2, thirdSep - (secondSep + 2));
00428           criticalOutputPort = this->moduleName_.substr(thirdSep + 2);
00429           logger_->debug("Extracted critical input port: {}", criticalInputPort);
00430           logger_->debug("Extracted output port: {}", criticalOutputPort);
00431         } else {
00432           logger_->warn("Can't find thirdSep. Invalid module name pattern: {}", this->moduleName_);
00433           return;
00434         }
00435       } else {
00436         logger_->warn("Can't find secondSep. Invalid module name pattern: {}", this->moduleName_);
00437         return;
00438       }
00439     } else {
00440       logger_->warn("Can't find firstSep. Invalid module name pattern: {}", this->moduleName_);
00441       return;
00442     }
00443   } else {
00444     logger_->warn("Module name is empty. Can't extract critical input port.");
00445     return;
00446   }
00447
00448   // Add additional wires for intermediate outputs that aren't part of the chain
00449   if (this->outputPins_.size() > 1) {               // Only add if there are multiple output pins
00450     for (int i = 0; i < instance_count_ - 1; i++) { // Skip the first and last instances
00451       for (const auto &outputPin : this->outputPins_) {
00452         if (outputPin != criticalOutputPort) { // Found an intermediate output pin
00453           auto &newNetNode = parse("\n  wire P_" + std::to_string(i) + "__" + outputPin + ";");
00454           insertAtBack(module.members, newNetNode);
00455           logger_->debug("Added intermediate wire: {}", newNetNode.toString());
00456         }
00457       }
00458     }
00459   }
00460
00461   std::string portList_str = module.header->ports->toString();
00462   logger_->debug("Port list: {}", portList_str);
00463
00464   std::vector<std::string> allPorts;
00465   auto ansiPortList = module.header->ports->as_if<slang::syntax::AnsiPortListSyntax>();
00466   if (!ansiPortList) {
00467     logger_->warn("Port list is not ANSI style.");
00468     return;
00469   }
00470   portInfoMap_.clear(); // Clear the port info map
```

```
00471   for (const auto portMember : ansiPortList->ports) {
00472     if (portMember->kind == slang::syntax::SyntaxKind::ImplicitAnsiPort) {
00473       const auto implicitPort = portMember->as_if<slang::syntax::ImplicitAnsiPortSyntax>();
00474       const auto &directionToken =
00475           implicitPort->header->as_if<slang::syntax::VariablePortHeaderSyntax>()->direction;
00476       const auto &nameToken = implicitPort->declarator->name;
00477
00478       std::string_view portName = nameToken.valueText();
00479       std::string_view direction = directionToken.valueText();
00480       portInfoMap_[std::string(portName)] = std::string(direction); // Store port info in the map
00481       logger_->debug("Port Name: {}, Direction: {}", portName, direction);
00482     } else {
00483       logger_->warn("Port member is not ImplicitAnsiPort.");
00484     }
00485   }
00486
00487   for (int i = 0; i < instance_count_; i++) {
00488     std::string instanceName = "I_" + this->cellName_ + "__X" + std::to_string(i) + "__" +
00489                                criticalInputPort + "__" + criticalOutputPort;
00490     std::string instanceCode = "\n  " + this->cellName_ + " " + instanceName + " (";
00491     std::vector<std::string> portConnections;
00492
00493     for (const auto &pair : portInfoMap_) {
00494       std::string portName = pair.first;
00495       std::string direction = pair.second;
00496       std::string connectionName;
00497
00498       if (portName == criticalInputPort) {
00499         if (i == 0) {
00500           connectionName = portName; // First instance: connect to module input port
00501         } else {
00502           connectionName =
00503               "OP_" + std::to_string(i - 1); // Intermediate instances: connect to previous OP wire
00504         }
00505       } else if (portName == criticalOutputPort) {
00506         if (i == instance_count_ - 1) {
00507           connectionName = portName; // Last instance: connect to module output port
00508         } else {
00509           connectionName = "OP_" + std::to_string(i); // Intermediate instances: connect to OP wire
00510         }
00511       } else if (direction == "output") { // Handle other output ports
00512         if (i < instance_count_ - 1) {
00513           connectionName = "P_" + std::to_string(i) + "__" +
00514                            portName; // Connect to intermediate P_i__portName wire
00515         } else {
00516           connectionName = portName; // Last instance: connect to module output port
00517         }
00518       } else { // For input ports (and inout?), connect directly to module port
00519         connectionName = portName;
00520       }
00521       portConnections.push_back("." + portName + "(" + connectionName + ")");
00522     }
00523
00524     // Connect all ports with comma and space
00525     if (!portConnections.empty()) {
00526       instanceCode += portConnections[0];
00527       for (size_t i = 1; i < portConnections.size(); ++i) {
00528         instanceCode += ", " + portConnections[i];
00529       }
00530     }
00531     instanceCode += ");";
00532
00533     // Blank line before the first instance
00534     if (i == 0) {
00535       instanceCode = "\n" + instanceCode;
00536     }
00537
00538     // Insert the instance code
00539     auto &instanceNode = parse(instanceCode);
```

```
00540     insertAtBack(module.members, instanceNode);
00541   }
00542 }
00543
00544 void getAST(const std::string &verilog_file, const std::string &cell) {
00545   try {
00546     spdlog::info("Starting get AST from Verilog file: '{}'", verilog_file);
00547     auto result = slang::syntax::SyntaxTree::fromFile(verilog_file);
00548     if (result) {
00549       spdlog::info("Successfully parsed Verilog file.");
00550
00551       try {
00552         // First, use the visitor to print basic information
00553         VerilogVisitor visitor(cell);
00554         visitor.visit(result.value()->root());
00555
00556         // If a target cell is specified, use the rewriter to extract the relevant code
00557         if (!cell.empty()) {
00558           // Create a rewriter to only keep the target cell related code
00559           CellExtractor extractor(cell);
00560           auto extractedTree = extractor.transform(result.value());
00561
00562           if (extractor.foundTargetCell()) {
00563             // Use SyntaxPrinter to print the extracted code
00564             std::string extractedCode = slang::syntax::SyntaxPrinter::printFile(*extractedTree);
00565
00566             // Save to a file named after the cell name
00567             std::string outputFile = cell + ".v";
00568             std::ofstream cellOut(outputFile);
00569             if (cellOut) {
00570               cellOut << extractedCode;
00571               cellOut.close();
00572               spdlog::info("Extracted '{}' cell code to '{}'", cell, outputFile);
00573             } else {
00574               spdlog::error("Failed to write extracted cell code to '{}'", outputFile);
00575             }
00576           } else {
00577             spdlog::warn("Target cell '{}' not found in the Verilog file", cell);
00578           }
00579         }
00580
00581         // spdlog::info("Print full source code to 'full_source_code.v'");
00582         // // Optionally save the entire syntax tree
00583         // std::string fullOutput = slang::syntax::SyntaxPrinter::printFile(*result.value());
00584         // std::ofstream out("full_source_code.v");
00585         // out << fullOutput;
00586         // out.close();
00587
00588         spdlog::info("Print target cell code to 'cell_code.v using toString()'");
00589         // Use CellPrinter to print the target cell's code
00590         std::ofstream cellOut("cell_code.v");
00591         CellPrinter cellPrinter(cell, cellOut);
00592         cellPrinter.visit(result.value()->root());
00593         cellOut.close();
00594
00595       } catch (const std::exception &e) {
00596         spdlog::error("Exception during AST traversal: {}", e.what());
00597       } catch (...) {
00598         spdlog::error("Unknown exception during AST traversal");
00599       }
00600     } else {
00601       spdlog::error("Error parsing Verilog file.");
00602     }
00603   } catch (const std::exception &e) {
00604     spdlog::error("Exception during Verilog parsing: {}", e.what());
00605   } catch (...) {
00606     spdlog::error("Unknown exception during Verilog parsing");
00607   }
00608 }
```

索引