TRISTAN eFPGA IP User Guide

YONGATEK MICROELECTRONICS

Ahmad Houraniah

TRISTAN eFPGA Project

The *TRISTAN eFPGA* project is an advanced initiative by YONGATEK Microelectronics aimed at designing a flexible, efficient, and customizable embedded FPGA (eFPGA) architecture. Leveraging the OpenFPGA framework, TRISTAN introduces innovative automation flows for synthesis, placement and routing, and bitstream generation. This guide details the tools, workflows, and scripts developed for TRISTAN to simplify FPGA design, configuration, and testing.

Contents

1 Introduction

The proposed eFPGA is constructed using the OpenFPGA framework. This document provides an introduction to OpenFPGA and outlines our custom toolkit designed to automate FPGA generation and customization. It also simplifies the user flow for bitstream generation and FPGA testing.

This user guide covers the OpenFPGA framework alongside our custom scripts and workflows. OpenFPGA is a powerful tool for generating and utilizing custom FPGAs. To enhance its capabilities and streamline the design process, we have developed a specialized toolset built on top of OpenFPGA.

1.1 General Flow of OpenFPGA

The general flow for using OpenFPGA is as follows:

1. Define the VPR architecture in XML format. Refer to the VPR documentation for details.
2. Define the OpenFPGA architecture in XML format. Refer to the OpenFPGA architecture documentation for details.
3. Define the synthesis script for Yosys.
4. Define the place-and-route (PnR) script or command (typically a single-line command).
5. Define the OpenFPGA shell script, which controls the behavior of the OpenFPGA flow.

Note: Synthesis is not controlled by the OpenFPGA shell. Therefore, synthesis commands must be executed prior to invoking the OpenFPGA flow.

2 OpenFPGA - Synthesis

Synthesis is handled by an open-source tool called **Yosys**. Yosys is suitable for FPGA synthesis; however, since the tool is not inherently aware of the specific hardware, we must provide files that describe the hardware characteristics. These include details such as flip-flop (FF) types, LUT sizes, and custom primitives.

The general flow for synthesis is described below. Note that this does not represent a complete synthesis script but provides the overall structure. Additional commands are required for a complete synthesis script.

```
read_verilog <benchmark>
read_verilog -lib specify <models for some of our FPGA primitives>

# BRAM Mapping
memory_libmap -lib <BRAM description file>
techmap -map <BRAM Verilog Map Model>
memory_map

# Multiplier Mapping
wreduce t:$mul
```

```
techmap -map +/mul2dsp.v -map <Mult Verilog Map Model> \
       -D DSP_SIGNEDONLY=1 \
       -D DSP_A_MAXWIDTH=7 \
       -D DSP_MAXWIDTH=7 \
       -D DSP_A_MINWIDTH=2 \
       -D DSP_NAME=mult_8x8 a:nodsp %n

# Adder Mapping
techmap -map +/techmap.v -map <Adder Verilog Mapping File>

# FF Map
dfflegalize -cell $_DFF_P_ 0
techmap -map +/adff2dff.v

# Map LUTs
abc -lut 4

# Generate Netlist
write_blif <file>
write_verilog <file>
```

Listing 1: Synthesis Flow

2.1 BRAM Mapping

BRAM Mapping requires two files, the BRAM description file and a BRAM Map Model. The description describes the type of memory available. For a true dual-port SRAM, the format is as follows:

```
ram block $BRAM_NAME {
    abits 10; // Address width
    width 8;  // Data width
    byte 8;   // Write enable behavior; current one writes to full line
    init none; // Initialization capabilities; if initializable,
               // the initial value would be represented as parameters
               // in the netlist which must be handled by the PnR flow.
    port srsw "A" "B" { // List of ports, each capable of srsw
        clock posedge;
    }
}
```

Listing 2: BRAM Description File

The BRAM Verilog Mapping file contains ports compatible with Yosys' internal representation. It is provided to map logic to the actual BRAM primitive.

2.2 Multiplier Mapping

`wreduce t:$mul` is used for multiplier optimization. The command maps multipliers as follows:

```
techmap -map +/mul2dsp.v -map <Mult Verilog Map Model> \
       -D DSP_SIGNEDONLY=1 \
       -D DSP_A_MAXWIDTH=7 \
       -D DSP_MAXWIDTH=7 \
       -D DSP_A_MINWIDTH=2 \
       -D DSP_NAME=mult_8x8 a:nodsp %n
```

Listing 3: Multiplier Mapping

Currently, all multiplications are mapped to signed 7x7 multipliers. This can be modified to perform unsigned 8x8 multiplications. Note that this does not cause hardware changes and can be adjusted in the user flow.

2.3 Adder Mapping

Hard adders included in CLBs are mapped using:

```
techmap -map +/techmap.v -map <Adder Verilog Mapping File>
```

Listing 4: Adder Mapping

2.4 Flip-Flop Mapping

The following commands define which FF types can be used in the netlist:

```
dfflegalize -cell $_DFF_P_ 0
techmap -map +/adff2dff.v
```

Listing 5: FF Mapping

2.5 Mapping LUTs

LUT mapping is handled with the following command:

```
abc -lut 4
```

Listing 6: LUT Mapping

2.6 Netlist Generation

The netlist can be generated in the following formats:

- `BLIF`: Accepted by VPR.
- `EBLIF`: Contains additional parameters, such as BRAM initialization, and is also accepted by VPR. However, BRAM initialization circuitry and programming logic must still be defined separately.

```
write_blif <file>
write_verilog <file>
```

Listing 7: Netlist Generation

3 OpenFPGA - Place and Route (PnR)

PnR is a critical step in the OpenFPGA flow, as some files generated by the PnR tool (`vpr`) are required to proceed with the flow.

3.1 Basic PnR Command

The most basic PnR command is:

```
vpr <xml_arch> <netlist_in_blif>
```

This command does not explicitly specify the FPGA size or the number of routing tracks. The tool attempts to determine the minimum size and track number required for the given benchmark (it keeps increasing both until the design is placed and routed). Therefore, this is not suitable for FPGA development, where the fabric is fixed.

3.2 Specifying Size and Tracks

To specify the FPGA size and number of tracks, use:

```
vpr <xml_arch> <netlist_in_blif> --device
<layout_name, e.g., 20x20> --route_chan_width 130
```

3.3 Additional Options

Additional PnR options include:

- Clock modeling (ideal or with routing delays).
- Benchmark SDC constraints.
- Other PnR strategies.

3.4 Post-PnR Steps

After completing PnR:

1. Read the OpenFPGA architecture. This step informs the tool about the primitives used in the hardware.
2. Run `build_fabric` to generate the FPGA fabric (without writing the RTL, which is generated internally).

3.5 Key Commands

- **Writing the fabric RTL:**

  ```
  write_fabric_verilog
  ```

- **Writing SDCs:**

  ```
  write_pnr_sdc
  ```

- **Generating the bitstream:**

```
build_architecture_bitstream
build_fabric_bitstream
```

- **Generating a testbench:**

  1. Generate the bitstream first.
  2. For a full testbench (includes programming):

     ```
     write_full_testbench
     ```

  3. For a preconfigured testbench:

     ```
     write_preconfigured_fabric_wrapper
     write_preconfigured_testbench
     ```

     Note: The preconfigured fabric wrapper creates a dummy wrapper for a configured FPGA using `force` statements.

4 FPGA Architecture Documentation

The FPGA architecture is defined in XML format. This consists of two main parts:

4.1 VPR Architecture

This is the main file where we define our FPGA. There are multiple subsections in this file that must be completed.

A. **Models:** These represent the FPGA primitives that are available. This is mainly used to define the ports for the primitives.

B. **Tiles:** This subsection defines the physical attributes for the tiles. A separate entry must be made for each tile type. We should also define the pin locations (top, bottom, right, left).

C. **Device:** We define the switch block type here and some other attributes.

D. **Switch List and Segment:** These define the different length switches/routing tracks that we have. The lengths specify the number of jumps for that switch.

E. **Direct List:** This defines the direct connections in the fabric, such as carry chains.

F. **Complex Block List:** This is the most complex part of the architecture. This subsection must define all the tiles or `pb_types`, describe their structure, and the connections between the tiles and their submodules in the hierarchy (e.g., how do LUT pins connect to FFs and the adder). This is also the subsection where we define the delays in the design, which are used to model the FPGA's expected performance.

4.2 OpenFPGA Architecture

A. **Technology Library:** This describes technology-specific features, but we have not made any changes to this subsection so far.

B. **Circuit Library:** In this subsection, we define the primitives and the cells that will be used to build the FPGA fabric (e.g., MUX, DFF).

C. **Configuration Protocol:** This is where we define the configuration protocol that will be used. We can also set the FF type to be used and the bitstream width here.

D. **Connection Block and Switch Block and Routing Segments:** Routing logic and circuit models for them.

E. **Direct Connection:** Connections that do not go through the routing network, such as carry chains.

F. **Tile Annotations:** Used for defining global pins.

G. **Pb_type Annotations:** Connects the models to the physical cells or primitives.

4.3 Design Variables

This file contains the delays that are used to model the FPGA's performance. These delays must be back-annotated from the implementation.

4.4 Clock and Reset Topology

In our architectures, we use global pins for both clocks and resets. This approach is chosen because so that CTS can be performed seperately, rather than relying on OpenFPGA's PnR tool (VPR).

4.4.1 Clock Domains

When using multiple clock domains in an architecture, we can utilize clock multiplexers to select the active clock for each Fracturable Logic Element (FLE). These multiplexers are automatically configured by OpenFPGA. Clocks are declared as global pins, ensuring they are not controlled by the routing logic. The current architecture supports 1 clock domain, but it can be extended to support multiple clock domains as well.

4.4.2 Reset Topology

The design includes a global asynchronous active-low reset input. While we recommend that users implement a logical synchronous reset for their designs, the global reset remains available for use. Declaring resets as global pins ensures they bypass the routing logic, allowing the global reset to directly reset all logical flip-flops (FFs) in the design.

5 Configuration Protocols

Configuration can be defined by multiple protocols. For more information, refer to the OpenFPGA documentation: `https://openfpga.readthedocs.io/en/master/manual/arch_lang/config_protocol/`

5.1 TRISTAN eFPGA Configuration Protocol

We are using the **configuration chain protocol**, which consists of scan chains throughout the FPGA. The flip-flops (FFs) along the scan chain control the configurable parts of the FPGA, such as the memory for look-up tables (LUTs) and the connections in the routing logic.

The width of the configuration chain, which is also the number of configuration chains in parallel, is adjustable by the user. We set this width to **32 bits**. Additionally, we have a `program_enable` signal to control when the data propagates through the scan chain. The `config_enable` signal should be pulsed for each input to the configuration chain.

The programming interface has the following ports:

Inputs

- `prog_clk`
- `config_en`
- `[31:0] ccff_head`

Outputs

- `[31:0] ccff_tail`

5.2 Notes

It is important to note that with the current structure, it is not possible to program the flip-flops (FFs) or BRAM memories. This limitation arises because the configuration chain is not connected to the logical FFs. BRAMs require specialized circuitry to be programmable. A potential approach could involve a counter-based mechanism to sequentially write data to all addresses of all BRAMs. However, this implementation is not currently available in the architecture. Moreover, no online examples or references have been found for this functionality. Further research is required to address BRAM initialization. Implementing a programmable mechanism for both FFs and BRAMs would enhance the configurability and flexibility of the architecture.

6 Simulation

OpenFPGA provides two options for simulating the bitstream. `https://openfpga.readthedocs.io/en/master/manual/fpga_verilog/testbench/`

6.1 Preconfigured Testbench

This approach skips the configuration stage by using a wrapper that applies `force` commands to all the configuration flops in the FPGA, forcing it to be in the programmed phase. This method bypasses the configuration logic, aiming to speed up the process. However, it does not scale linearly with the FPGA's complexity and, therefore, does not necessarily reduce the simulation time significantly compared to the full testbench that programs the FPGA first.

6.2 Full Testbench

The full testbench reads the data from the bitstream and writes it to the FPGA through the configuration logic.

6.3 Post-Configuration Testing

After the configuration is complete, both testbenches apply the same methodology. The approach involves performing a basic test of logical equivalency between the programmed FPGA and the benchmark. This is achieved by providing random input stimulus to the FPGA and then comparing the outputs between the benchmark's RTL and the programmed FPGA.

6.4 Custom Testbench

Using the preconfigured wrapper, we can apply any testbench to the FPGA. This is specifically useful for testbenches where random stimulus is not sufficient to verify functionality. With this method, the user has to provide their own testbench and instanitate the programmed FPGA (preconfigured wrapper) as the unit under test.

7 Defining a Task in OpenFPGA

To define a task to run in OpenFPGA, we require a `task.conf` file.
In this file, we specify:

- The flow to run,
- The VPR architecture,
- The benchmark,
- The OpenFPGA shell script,
- Any additional commands to pass to synthesis or the OpenFPGA shell script.

The `task.conf` file consists of the following sections:

7.0.1 [GENERAL]

Description: Define general information about this task.

7.0.2 [OpenFPGA SHELL]

Description:

- Specify the OpenFPGA shell script to use.
- Define the commands or arguments to pass to the OpenFPGA shell script and the PnR script.
- Set the hardware specifications, such as:

- FPGA size,
- Layout name,
- Number of routing tracks.

7.0.3 [ARCHITECTURES]

Description: Define the VPR architecture here.

7.0.4 [BENCHMARKS]

Description:

- Define the path for the benchmark.
- Note that VHDL is not fully supported at the moment.
- To support VHDL, customize the synthesis script and use the GHDL plugin.

7.0.5 [SYNTHESIS_PARAM]

Description:

- Define the synthesis script and the arguments to pass to it.
- Include the benchmark's top module name.
- Specify the pin constraints files.

8 Pin Constraints

Defining appropriate pin constraints requires several steps.

OpenFPGA by default flattens all pins for a benchmark. For example,

$$A[2:0] \rightarrow A\_0\_, A\_1\_, A\_2\_$$

after running the FPGA flow. This can cause issues when trying to run simulations, as the benchmark itself is not flattened. This is dealt with by defining a `bus_group.xml` file, which defines these bus groups, allowing OpenFPGA to adjust the flow, specifically the testbench generation part, to form the groups again.

To define clock and reset pins (or any other global pins), we define a `pin_constraints.xml` file, which has a structure as shown below:

```
<pin_constraints>
 <set_io pin="clk[0]" net="clk"/>
</pin_constraints>
```

This maps the global pin `clk[0]` to the benchmark's `clk` pin. Due to a limitation from OpenFPGA, these global pins will still be mapped to GPIOs, however, they will only be used through the global pins.

To map a benchmark's ports to specific GPIOs, we require multiple files:

1. `fpga_io_location.xml`: This file can be generated from OpenFPGA, and it represents the physical location in terms of x, y, z coordinates, where x is the horizontal coordinate, y is the vertical coordinate, and z is the pin number for a tile. x and y start from the bottom left. For example, coordinates {0, 1, 3} represent pin number 3 in tile {0, 1}.
2. `pin_map.csv`: This file must be written after any fabric is generated. It contains information about the orientation, pin name, mapping name, and direction.
3. `constraints.pcf`: This file is where the user sets the pin constraints for the FPGA. The syntax is as follows:

```
set_io <benchmark pin name> <target fpga pin>
```

Note that step 2 is now automatically generated by one of our scripts, which will be described in the document later.

9 eFPGA Flow

Due to the various parts that require modifications to run the flows, we have developed an automated script and flow, **eFPGA Flow**, that simplifies the process, making OpenFPGA suitable for use without requiring too much OpenFPGA-specific knowledge.

9.1 Task Selection

We offer 6 specific flows under the **eFPGA Flow**:

1. Generate Fabric
2. Generate SDC
3. Preconfigured testbench
4. Custom Testbench
5. Full Testbench
6. Generate Bitstream

If the task selected is from options 3-6, the script will prompt the user to select a benchmark from a list of benchmarks found in a directory. This directory is under `efpga-design-flow/benchmarks` in the current repository structure. The user will then be prompted to set a pin constraints file. If the user selects `no`, the pin assignment is left to VPR, which produces mostly random results. Otherwise, if a matching file (same name as the benchmark) is found under `efpga-design-flow/pin_constraints`, this file is used as is. If no matching file is found, then the script will constrain the pins by the order of appearance in the RTL. After this step, the script will automatically update the task configuration file (`task.conf`) and run the OpenFPGA flow. If options 3-5 are selected, the script will then automatically run the simulation using the selected simulator (which is currently Synopsys VCS). The script also records the results from the OpenFPGA (area and timing) run and the simulation result. Additionally, the script allows the user to run a task type for all available benchmarks. This is useful for a performance evaluation of the FPGA architecture.

10 eFPGA Flow User Guide

10.1 Introduction

This guide provides instructions for using the eFPGA Flow for FPGA design automation with OpenFPGA. The scripts in this repository are suitable for both Docker and standalone installations of OpenFPGA.

10.2 Installation

Clone the repository and navigate to the configuration directory:

```
cd efpga-design-flow/fpga_flow/
```

This directory is used to run all FPGA flow tasks, including RTL and SDC generation, simulation, and bitstream generation.

10.3 Running the Script

There is a Python script named `fpga_task.py`, which automates several parts of the design. To run the script, use:

```
python3 fpga_task.py
```

This will prompt the user to select the specific flow that they want to run.

10.4 Benchmark Selection and Addition

When you run the `fpga_task.py` script, you get an option to select a benchmark. This list is based on all the Verilog files in `efpga-design-flow/benchmarks/`. If a design has multiple files, then they should be placed in a new folder inside `efpga-design-flow/benchmarks/`.

Notes:

- The benchmark top module must match the file name (e.g., if the top module name is `or2`, then the file name should be `or2.v`).
- If the design contains multiple files, they should be placed in a folder where the folder has the same name as the top module (e.g., if the top module name is `turbo`, then `efpga-design-flow//benchmarks/turbo/` should contain all the files).
- When selecting a custom testbench option for the flow, the testbench should be in this directory as well, with the same name followed by `_tb` (e.g., `or2.v` and `or2_tb.v`).

The reason for these requirements is that our script needs to be able to identify the top module name without requiring manual input. This naming scheme simplifies the process, making it faster for the user to test different benchmarks quickly.

10.5 Synthesis Files

The synthesis script and other synthesis files can be found under:

`efpga-design-flow/run_fpga_task/config/yosys_dep/`

The OpenFPGA shell script, which contains PnR commands, can be found in:

`efpga-design-flow/yonga_openfpga_shell_scripts`

10.6 Benchmarks and Constraints

Benchmarks can be found in:

`efpga-design-flow/benchmarks`

Pin constraints can be found in:

`efpga-design-flow/pin_constraints`

Simulation settings are configured in:

`efpga-design-flow/misc/fixed_sim_openfpga.xml`

10.7 Task Configuration

Task configuration settings are defined in the file:

`efpga-design-flow/fpga_flow/config/task.conf`

This file allows you to change the FPGA size and layout, selecting a layout from those defined in the XML file, and the number of routing tracks.

10.8 Run Outputs

The specific runs will be generated in the following directory:

`efpga-design-flow/fpga_flow/run###`

Where is the run number. To view the results of a specific run, navigate to:

`efpga-design-flow/fpga_flow/run###/<benchmark_name>/MIN_ROUTE_CHAN_WIDTH/`

In this directory, you can find the following logs:

- `yosys_output.log` (Synthesis log)
- `vpr_stdout.log` (PnR log)
- `openfpgashell.log` (OpenFPGA log)

The generated bitstream, if it exists, will be in this directory, named `fabric_bitstream.bit`.

10.9 Testbench and Simulation

If a testbench is generated, the testbench and simulation modules will be located in:

`SRC/`

The benchmark is also copied to this directory for simulation, which can be found under:

`benchmark/`

10.10 Debugging Notes

11 Demo Flow

The following steps outline a demo run using the eFPGA Flow:

1. Navigate to the installation path of the repository:

   ```
   cd <installation path>/efpga-design-flow
   ```

2. Move to the fpga task directory:

   ```
   cd fpga_task
   ```

3. Run the task script:

   ```
   python3 fpga_task.py
   ```

4. Select an option from the prompt:

   ```
   "
   0: Generate_fabric
   1: Generate_sdc
   2: Preconfigured testbench
   3: Custom preconfigured testbench (when available)
   4: Full testbench
   5: Generate Bitstream
   "
   ```

   For example, selecting `0` will generate the fabric.

   ```
   Fabric is now generated, RTL can be found in

   efpga-design-flow/yonga_archs/Fabric/SRC
   ```

5. Run the script again to generate SDC files:

```
python3 fpga_task.py
```

   Select `1`:

```
SDCs are now generated, they can be found in

efpga-design-flow/yonga_archs/Fabric/SDC

and efpga-design-flow/yonga_archs/Fabric/tile_SDC
```

6. To execute a full testbench:

```
python3 fpga_task.py
```

   Select `4` and then choose a benchmark from the list:

```
"
0: add_sub_8bit.v
1: counter.v
2: or2.v
...
N: <last benchmark>
"
```

   Benchmarks are sorted alphabetically. For example, selecting `2` corresponds to the `or2` benchmark.

7. Specify whether to set a PCF (Pin Constraints File):

```
Set PCF? (y/n):
```

   Choosing `y` constrains pins according to a specified PCF, while `n` allows semi-random mapping of pins. Read section 8 for a more detailed explanation on PCF files.

8. The simulation is executed using the selected benchmark. Results, benchmark utilization, and frequency are displayed in the terminal. Figure 1 shows the expected output.

9. VCD waveforms are generated and saved in the following directory:

```
../latest/<arch name>/OR2/MIN_ROUTE_CHAN_WIDTH/or2_formal.vcd
```

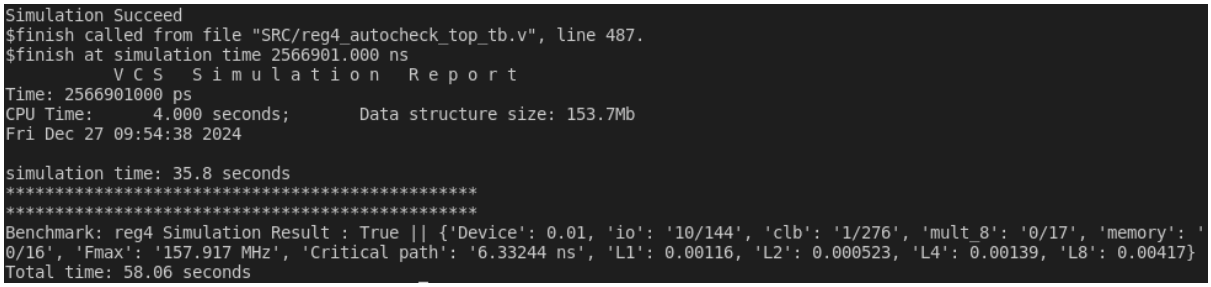

Figure 1: Task Output

12 FPGA Task Automation Script

12.1 Overview

This script automates the process of generating FPGA fabrics, generating bitstreams and testbenches, evaluating FPGA performance, and running simulations. It integrates various utilities and parsers to streamline the FPGA design flow.

12.2 Modules and Functions

12.2.1 Main Script

The main script (`fpga_task.py`) orchestrates the entire flow. It initializes the task configuration, selects benchmarks, sets the simulator, and runs tasks for each benchmark.

- `use_docker()`: Checks if the script is running in a Docker environment.
- `get_benchmarks_and_tbs()`: Retrieves and sorts the list of benchmarks and testbenches.
- `select_benchmark(benchmarks, tb_type)`: Selects a benchmark based on user input.
- `run_task_for_benchmark(task_conf_script, benchmark, tb_type, arch_name, simulator, tbs, autoset_pcf)`: Runs the task for a given benchmark.
- `set_simulator()`: Sets the simulator to use for simulation.
- `main()`: The main function that initializes the task configuration, selects benchmarks, sets the simulator, and runs tasks for each benchmark.

12.2.2 Task Configuration

The `TaskConf` class in `task_conf.py` handles the configuration of the FPGA task. It reads the configuration file, sets the benchmark, updates the task configuration, and runs the task.

- `__init__(self, is_docker, pins_per_io)`: Initializes the TaskConf object.
- `_read_config(self)`: Reads the task configuration file.
- `get_fpga_size(self)`: Returns the FPGA size.

- `get_arch_name(self)`: Returns the architecture name.
- `set_benchmark(self, benchmark)`: Sets the benchmark object.
- `set_openfpga_shell_script(self, tb_type)`: Sets the appropriate OpenFPGA shell script based on testbench type.
- `update_task_conf(self)`: Updates the task configuration file with the benchmark's name/path and pin constraints.
- `run_task(self)`: Runs the task based on the testbench type.

12.2.3 Simulation

The `simulate.py` script handles the simulation of the FPGA design. It generates the simulation command based on the simulator type, testbench type, and availability of custom testbench.

- `generate_simulation_command(simulator, tb_type, benchmark, tbs, task_dir)`: Generates the simulation command.
- `simulate(tb_type, benchmark, arch_name, simulator, tbs, run_status)`: Runs the simulation and extracts results.

12.2.4 Utilities and Parsers

Various utility scripts and parsers are used to handle specific tasks in the FPGA design flow.

- `verilator_component_parser.py`: Parses and dissects Verilog/SystemVerilog source files for Verilator.
- `include_parser.py`: Modifies include statements in Verilog files.
- `flist_parser.py`: Updates fabric file lists based on Verilator Component Parser modifications.
- `fix_tb.py`: Fixes issues in autogenerated testbenches.
- `fix_sdc.py`: Generates and modifies Synopsys Design Constraints (SDC) files.
- `fix_netlist.py`: Fixes issues in the generated netlist files.
- `benchmark.py`: Stores information about the current benchmark.
- `paths.py`: Provides helper functions to get paths of the efpga-design-flow directory.
- `generate_layout_csv.py`: Generates the FPGA CSV layout file based on the generated RTL.

12.3 Execution Flow

The script starts by initializing the task configuration and selecting benchmarks. It then sets the simulator and runs tasks for each benchmark. Depending on the testbench type, it generates the necessary files, runs simulations, and extracts results. The results are then saved to a CSV file.

12.4 Usage

To run the script, execute the `fpga_task.py` file. The script will prompt for the testbench type and benchmark selection. It will then proceed to run the tasks and simulations, and output the results.

```
python3 fpga_task.py
```