BAG Documentation

Release 2.0

Eric Chang

Jun 21, 2019

Contents

Contents:

CHAPTER 1

Tutorial

This section contains several simple tutorials for you to get an idea of the BAG workflow.

In these tutorials, we will be using **`git`** extensively. git allows you to copy a working setup, and it also allows you to checkout and use other people's design while they can work on adding future improvements. To learn git, you can read the documentations here, or alternatively you can just google git commands to learn more about it while working through the tutorial.

CHAPTER 2

Overview

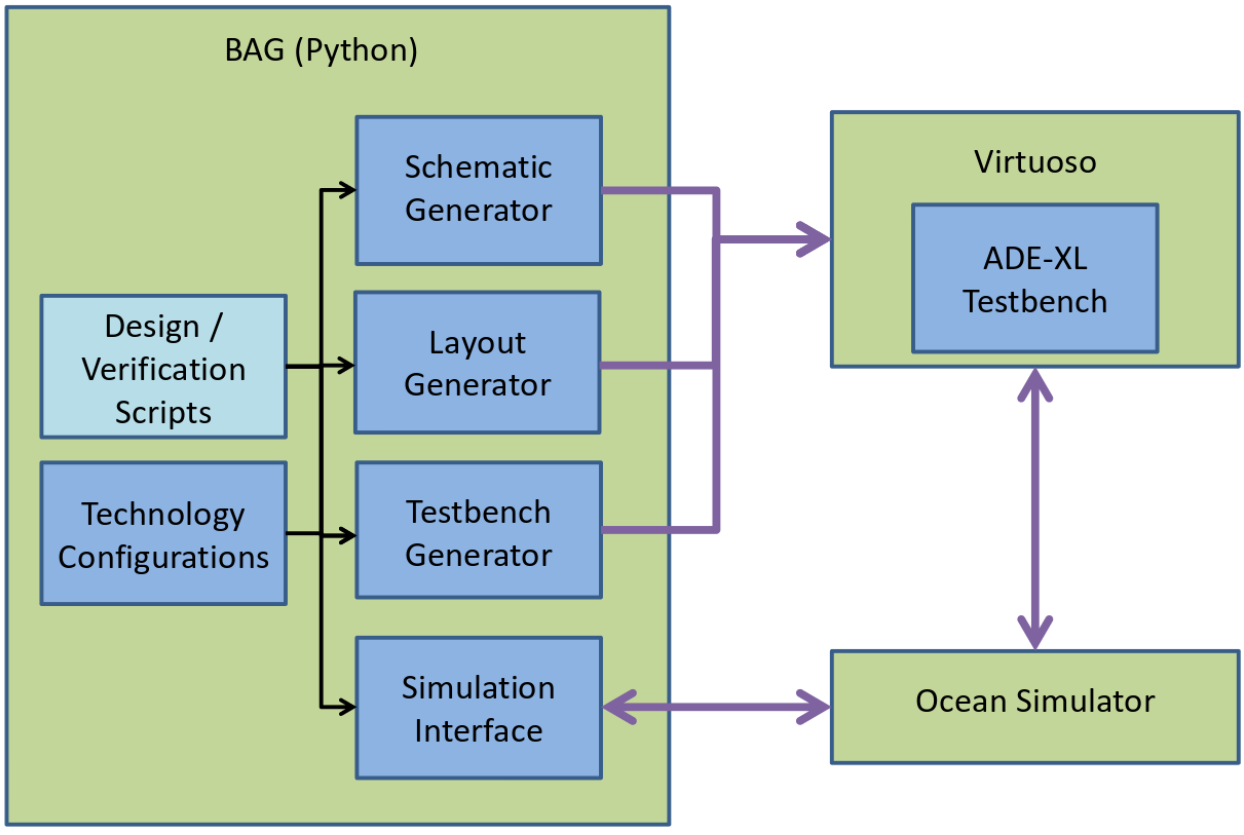

Fig. 1: BAG design flow diagram

BAG is a Python-based circuit design platform that aims to automate analog circuit design, but at the same time give the user full visibility and control over every step in the design flow.

The analog circuit design flow is generally as follows:

1. Create a schematic generator of the circuit.
2. Create a testbench generator to measure specifications and verify functionality.
3. Create a layout generator if post-extraction verification is needed.
4. Generate a schematic with given specifications.
5. Generate a testbench that instantiates the generated schematic.
6. Simulate the testbenches and post-process data to verify that the circuit meets specifications.
7. Create the layout of your schematic and verify it's LVS/DRC clean.
8. Repeat step 3 on post-extraction schematic.

BAG 2.0 is designed so that any or all steps of the design flow can be performed in a Python script or console, thus enabling rapid design iteration and architecture exploration.

To achieve its goal, BAG is divided into 4 components: schematic generators, layout generators, design modules, and testbench generators. These components are independent from one another, so the designer can pick and choose which steps in the design flow to automate. For example, the designer can simply use BAG to generate new schematics, and use his own CAD program for simulation and verification. Alternatively, The designer can provide an existing schematic to BAG and simply use it to automate the verification process.

BAG interacts with an external CAD program or simulator to complete all the design and simulation tasks. BAG comes with Virtuoso and Ocean simulator support, but can be extended to other CAD programs or simulators. The rest of this document assumes you are using Virtuoso and running simulations in Ocean.

Next we will describe each components of BAG in detail.

2.1 Schematic Generator

A schematic generator is a schematic in your CAD program that tells BAG all the information needed to create a design. BAG creates design modules from schematic generators, and BAG will copy and modify schematic generators to implement new designs.

A schematic generator needs to follow some rules to work with BAG:

1. Instances in a schematic generator must be other schematic generators, or a cell in the `BAG_prim` library.
2. BAG can array any instance in a schematic generator. That is, in the design implementation phase, BAG can copy/paste this instance any number of times, and modify the connections or parameters of any copy. This is useful in creating array structures, such as an inverter chain with variable number of stages, or a DAC with variable number of bits.

   However, if you need to array an instance, its ports must be connected to wire stubs, with net labels on each of the wire stubs. Also, there must be absolutely nothing to the right of the instance, since BAG will array the instance by copying-and-pasting to the right. An example of an inverter buffer chain schematic generator is shown below.

3. BAG can replace the instance master of any instance. The primary use of this is to allow the designer to change transistor threshold values, but this could be used for other schematic generators if implemented. Whenever you switch the instance master of an instance, the symbol of the new instance must exactly match the old instance, including the port names.

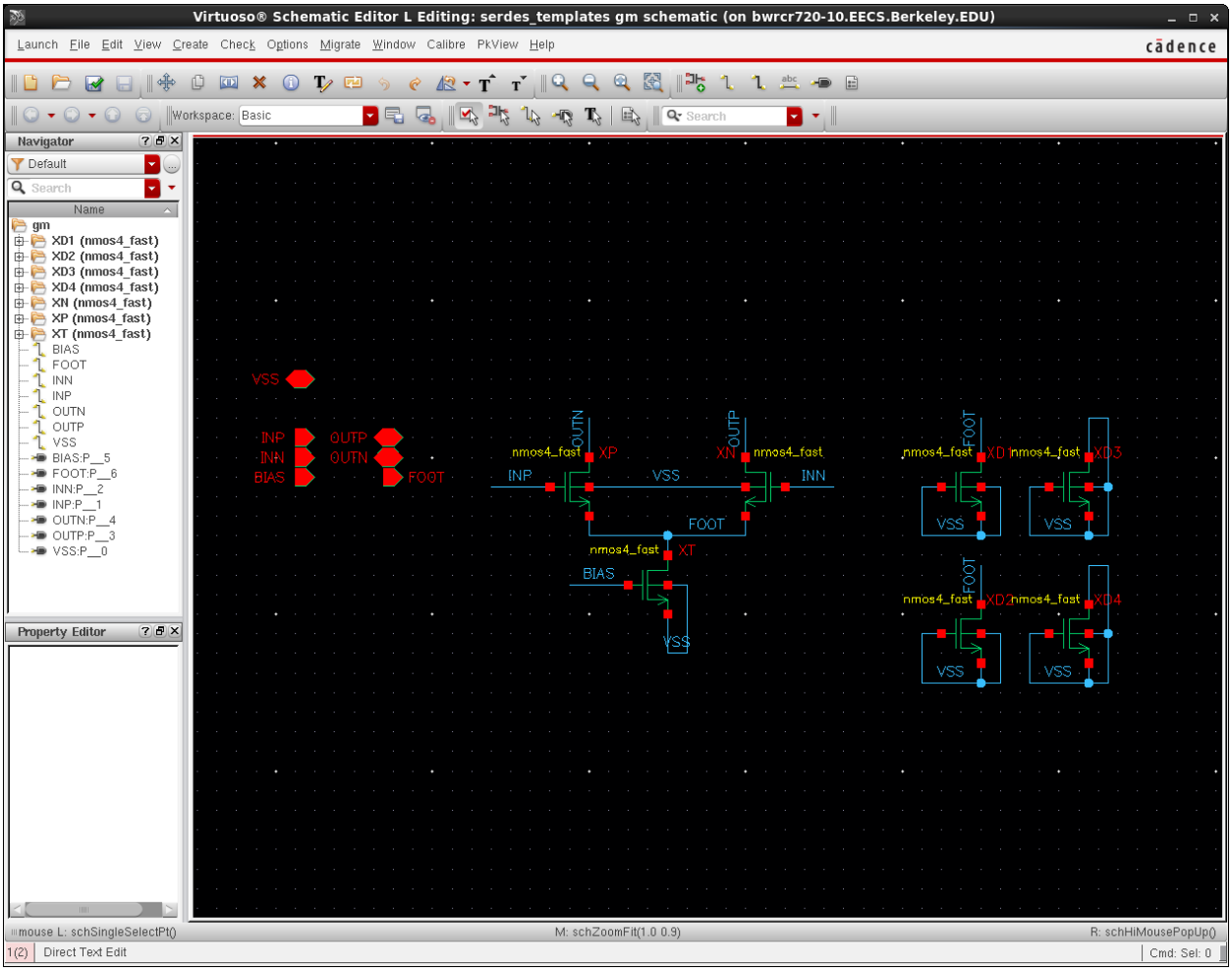

Fig. 2: An example schematic generator of a differential gm cell.

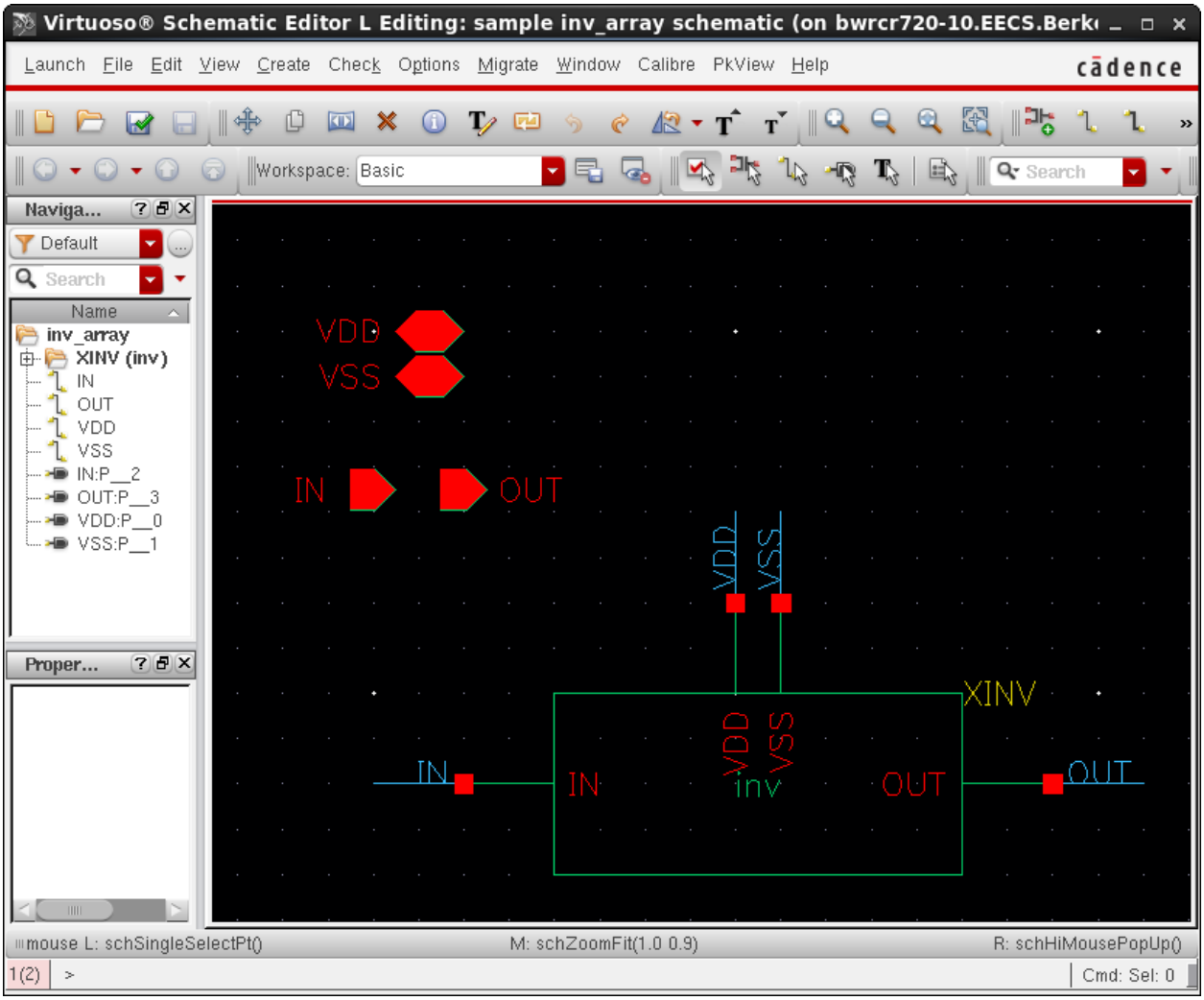

Fig. 3: An example schematic generator of an inverter buffer chain. Ports connected by wire stubs, nothing on the right.

4. Although not required, it is good practice to fill in default parameter values for all instances from the `BAG_prim` library. This makes it so that you can simulate a schematic generator in a normal testbench, and make debugging easier.

2.2 Design Module

A design module is a Python class that generates new schematics. It computes all parameters needed to generate a schematic from user defined specifications. For example, a design module for an inverter needs to compute the width, length, and threshold flavor of the NMOS and PMOS to generate a new inverter schematic. The designer of this module can let the user specify these parameters directly, or alternatively compute them from higher level specifications, such as fanout, input capacitance, and leakage specs.

To create a default design module for a schematic generator, create a `BagProject` instance and call `import_design_library()` to import all schematic generators in a library from your CAD program into Python. The designer should then implement the three methods, `design()`, `get_layout_params()`, and `get_layout_pin_mapping()` (The latter two are optional if you do not use BAG to generate layout). Once you finish the design module definition, you can create new design module instances by calling `create_design_module()`.

The following sections describe how each of these methods should be implemented.

2.2.1 design()

This method computes all parameters needed to generate a schematic from user defined specifications. The input arguments should also be specified in this method.

A design module can have multiple design methods, as long as they have difference names. For example, You can implement the `design()` method to compute parameters from high level specifications, and define a new method named `design_override()` that allows the user to assign parameter values directly for debugging purposes.

To enable hierarchical design, design module has a dictionary, `instances`, that maps children instance names to corresponding design modules, so you can simply call their `design()` methods to set their parameters. See *Tutorial* for an simple example.

If you need to modify the schematic structure (such as adding more inverter buffers), you should call the corresponding methods before calling `design()` methods of child instances, as those design module could be changed. The rest of this section explains how you modify the schematic.

Pin Renaming

Most of the time, you should not rename the pin of schematic. The only time you should rename the pin is when you have a variable bus pin where the number of bits in the bus can change with the design. In this case, call `rename_pin()` to change the number of bits in the bus. To connect/remove instances from the added/deleted bus pins, see *Instance Connection Modification*

Delete Instances

Delete a child instance by calling `delete_instance()`. After this call, the corresponding value in `instances` dictionary will become `None`.

Note: You don't have to delete 0-width or 0-finger transistors; BAG already handles that for you.

Replace Instance Master

If you have two different designs of a child instance, and you want to swap between the two designs, you can call `replace_instance_master()` to change the instance master of a child.

Note: You can replace instance masters only if the two instance masters have exactly the symbol, including pin names.

Instance Connection Modification

Call `reconnect_instance_terminal()` to change a child instance's connection.

Arraying Child Instances

Call `array_instance()` to array a child instance. After this call, `instances` will map the child instance name to a list of design modules, one for each instance in the array. You can then iterate through this list and design each of the instances. They do not need to have the same parameter values.

Restoring to Default

If you are using the design module in a design iteration loop, or you're using BAG interactively through the Python console, and you want to restore a deleted/replaced/arrayed child instance to the default state, you can call `restore_instance()`.

2.2.2 get_layout_params()

This method should return a dictionary from layout parameter names to their values. This dictionary is used to create a layout cell that will pass LVS against the generated schematic.

2.2.3 get_layout_pin_mapping()

This method should return a dictionary from layout pin names to schematic pin names. This method exists because a layout cell may not have the same pin names as the schematic. If a layout pin should be left un-exported, its corresponding value in the dictionary must be `None`.

This dictionary only need to list the layout pins that needs to be renamed. If no renaming is necessary, an empty dictionary can be returned.

2.3 Testbench Generator

A testbench generator is just a normal testbench with schematic and adexl view. BAG will simply copy the schematic and adexl view, and replace the device under test with the new generated schematic. There are only 3 restrictions to the testbench:

1. All device-under-test's (DUTs) in the testbench must have an instance name starting with `XDUT`. This is to inform BAG which child instances should be replaced.

2. The testbench must be configured to simulate with ADE-XL. This is to make parametric/corner sweeps and monte carlo easier.
3. You should not define any process corners in the ADE-XL state, as BAG will load them for you. This makes it possible to use the same testbench generator across different technologies.

To verify a new design, call `create_testbench()` and specify the testbench generator library/cell, DUT library/cell, and the library to create the new testbench in. BAG will create a `Testbench` object to represent this testbench. You can then call its methods to set the parameters, process corners, or enable parametric sweeps. When you're done, call `update_testbench()` to commit the changes to Virtuoso. If you do not wish to run simulation in BAG, you can then open this testbench in Virtuoso and simulate it there.

If you want to start simulation from BAG and load simulation data, you need to call `add_output()` method to specify which outputs to record and send back to Python. Output expression is a Virtuoso calculator expression. Then, call `run_simulation()` to start a simulation run. During the simulation, you can press `Ctrl-C` anytime to abort simulation. When the simulation finish, the result directory will be saved to the attribute `save_dir`, and you can call `bag.data.load_sim_results()` to load the result in Python. See *Tutorial* for an example.

Since BAG uses the ADE-XL interface to run simulation, all simulation runs will be recorded in ADE-XL's history tab, so you can plot them in Virtuoso later for debugging purposes. By default, all simulation runs from BAG has the `BagSim` history tag, but you can also specify your own tag name when you call `run_simulation()`. Read ADE-XL documentation if you want to know more about ADE-XL's history feature.

CHAPTER 3

BAG Setup Procedure

This document describes how to install Python for BAG and the various configuration settings. Since a lot of the configuration depends on the external CAD program and simulator, this document assumes you are using Virtuoso and Ocean (with ADEXL) for schematic design and simulation, respectively.

3.1 Installing Python for BAG

This section describes how to install Python for running BAG.

3.1.1 Installation Requirements

BAG is compatible with Python 3.5+ (Python 2.7+ is theoretically supported but untested), so you will need to have Python 3.5+ installed. For Linux/Unix systems, it is recommended to install a separate Python distribution from the system Python.

BAG requires multiple Python packages, some of which requires compiling C++/C/Fortran extensions. Therefore, it is strongly recommended to download Anaconda Python, which provides a Python distribution with most of the packages preinstalled. Otherwise, please refer to documentation for each required package for how to install/build from source.

3.1.2 Required Packages

In addition to the default packages that come with Anaconda (numpy, scipy, etc.), you'll need the following additional packages:

- python-future

  This package provides Python 2/3 compatibility. It is installable from `pip`:

  ```
  > pip install future
  ```

  `pip` also works for pre-downloaded tar file:

```
> pip install future-0.16.0.tar.gz
```

- subprocess32 (Python 2 only)

  This package is a backport of Python 3.2's subprocess module to Python 2. It is installable from `pip`.

- sqlitedict

  This is a dependency of OpenMDAO. It is installable from `pip`.

- OpenMDAO

  This is a flexible optimization framework in Python developed by NASA. It is installable from `pip`.

- mpich2 (optional)

  This is the Message Passing Interface (MPI) library. OpenMDAO and Pyoptsparse can optionally use this library for parallel computing. You can install this package with:

```
> conda install mpich2
```

- mpi4py (optional)

  This is the Python wrapper of `mpich2`. You can install this package with:

```
> conda install mpi4py
```

- ipopt (optional)

  Ipopt is a free software package for large-scale nonlinear optimization. This can be used to replace the default optimization solver that comes with scipy. You can install this package with:

```
> conda install --channel pkerichang ipopt
```

- pyoptsparse (optional)

  `pyoptsparse` is a python package that contains a collection of optmization solvers, including a Python wrapper around `Ipopt`. You can install this package with:

```
> conda install --channel pkerichang pyoptsparse
```

3.2 Building Pyoptsparse

To be written.

3.3 Configuration Files Summary

Although BAG has many configuration settings, most of them do not need to be changed. This file summarizes which settings you should modify under various use cases.

3.3.1 Starting New Project

For every new project, it is a good practice to keep a set of global configuration files to make sure everyone working on the project is simulating the same corners, running LVS and extraction with the same settings, and so on. In this case, you should change the following fields to point to the global configuration files:

- *database.testbench.env_file*
- *database.checker.lvs_runset*
- *database.checker.rcx_runset*
- *database.calibreview.cell_map*

3.3.2 Customizing Virtuoso Setups

If you changed your Virtuoso setup (configuration files, working directory, etc.), double check the following fields to see if they need to be modified:

- *database.checker.lvs_run_dir*
- *database.checker.rcx_run_dir*
- *simulation.init_file*

3.3.3 Python Design Module Customization

The following fields control how BAG 2.0 finds design modules, and also where it puts new imported modules:

- *lib_defs*
- *new_lib_path*

3.3.4 Changing Process Technology

If you want to change the process technology, double check the following fields:

- *database.schematic.tech_lib*
- *database.schematic.exclude_libraries*
- *database.testbench.config_libs*
- *class*

The following fields probably won't change, but if something doesn't work it's worth to double check:

- *database.schematic.sympin*
- *database.schematic.ipin*
- *database.schematic.opin*
- *database.schematic.iopin*
- *database.schematic.simulators*

3.4 BAG Configuration File

BAG configuration file is written in YAML format. This document describes each setting. BAG configuration file may use environment variable to specify values of any entries.

3.4.1 socket

This entry defines socket settings for BAG to communicate with Virtuoso.

socket.host

The host of the BAG server socket, i.e. the machine running the Virtuoso program. usually `localhost`.

socket.port_file

File containing socket port number for BAG server. When Virtuoso starts the BAG server process, it finds a open port and bind the server to this port. It then creates a file with name in `$BAG_WORK_DIR` directory, and write the port number to this file.

socket.sim_port_file

File containing socket port number for simulation server. When the simulation server starts, it finds a open port and bind the server to this port. It then creates a file with name in `$BAG_WORK_DIR` directory, and write the port number to this file.

socket.log_file

Socket communication debugging log file. All messages sent or received by BAG will be recorded in this log.

socket.pipeline

number of messages allowed in the ZMQ pipeline. Usually you don't have to change this.

3.4.2 database

This entry defines all settings related to Virtuoso.

data.class

The Python class that handles database interaction. This entry is mainly to support non-Virtuoso CAD programs. If you use Virtuoso, the value must be `bag.interface.skill.SkillInterface`.

database.schematic

This entry contains all settings needed to read/generate schematics.

database.schematic.tech_lib

Technology library. When BAG create new libraries, they will be attached to this technology library. Usually this is the PDK library provided by the foundry.

database.schematic.sympin

Instance master of symbol pins. This is a list of library/cell/view names. Most of the time this should be `["basic", "sympin", "symbolNN"]`.

database.schematic.ipin

Instance master of input pins in schematic. This is a list of library/cell/view names. Most of the time this should be `["basic", "ipin", "symbol"]`.

database.schematic.opin

Instance master of output pins in schematic. This is a list of library/cell/view names. Most of the time this should be `["basic", "opin", "symbol"]`.

database.schematic.iopin

Instance master of inout pins in schematic. This is a list of library/cell/view names. Most of the time this should be `["basic", "iopin", "symbolr"]`.

database.schematic.simulators

A list of simulators where the `termOrder` CDF field should be defined.

When Virtuoso convert schematics to netlists, it uses the `termOrder` CDF field to decide how to order the pin names in the netlist. This entry makes BAG update the `termOrder` field correctly whenever pins are changed.

Most of the time, this should be `["auLvs", "auCdl", "spectre", "hspiceD"]`.

database.schematic.exclude_libraries

A list of libraries to exclude when importing schematic generators to BAG. Most of the time, this should be `["analogLib", "basic", {PDK}]`, where `{PDK}` is the PDK library.

database.testbench

This entry contains all settings needed to create new testbenches.

database.testbench.config_libs

A string of config view global libries, separated by spaces. Used to generate config view.

database.testbench.config_views

A string of config view global cellviews, separated by spaces. Used to generate config view. Most of the time this should be `"spectre calibre schematic veriloga"`.

database.testbench.config_stops

A string of config view global stop cellviews, separated by spaces. Used to generate config view. Most of the time this should be `"spectre veriloga"`.

database.testbench.env_file

The simulation environment file name. A simulation environment is a combination of process corner and temperature. For example, if you simulate your circuit at TT corner with a temperature of 50 degrees Celsius, you may say the simulation environment is TT_50. A simulation environment file contains all simulation environments you want to define when BAG creates a new testbench. This file can be generated by exporting corner setup from an ADE-XL view.

database.testbench.def_files

A list of ADE/spectre definition files to include. Sometimes, a process technology uses definition files in addition to model files. If so, you can specify definition files to include here as a list of strings. Use an empty list (`[]`) if no definition file is needed.

database.testbench.default_env

The default simulation environment name. See *database.testbench.env_file*.

database.checker

This entry contains all settings needed to run LVS/RCX from BAG.

database.checker.checker_cls

The Python class that handles LVS/RCX. If you use Calibre with Virtuoso for LVS/RCX, the value must be `bag.verification.calibre.Calibre`.

database.checker.lvs_run_dir

LVS run directory.

database.checker.rcx_run_dir

RCX run directory

database.checker.lvs_runset

LVS runset.

database.checker.rcx_runset

RCX runset.

database.checker.source_added_file

Location of the source.added file for Calibre LVS. If this entry is not defined, BAG defaults to `$DK/Calibre/lvs/source.added`.

database.checker.rcx_mode

Whether to use Calibre PEX or Calibre XACT3D flow to perform parasitic extraction. The value should be either `pex` or `xact`. If this entry is not defined, BAG defaults to `pex`.

database.checker.xact_rules

Location of the Calibre XACT3D rules file. This entry must be defined if using Calibre XACT3D flow.

database.calibreview

This entry contains all settings needed to generate calibre view after RCX.

database.calibreview.cell_map

The calibre view cellmap file.

database.calibreview.view_name

view name for calibre view. Usually `calibre`.

3.4.3 simulation

This entry defines all settings related to Ocean.

simulation.class

The Python class that handles simulator interaction. This entry is mainly to support non-Ocean simulators. If you use Ocean, the value must be `bag.interface.ocean.OceanInterface`.

simulation.prompt

The ocean prompt string.

simulation.init_file

This file will be loaded when Ocean first started up. This allows you to configure the Ocean simulator. If you do not want to load an initialization file, set this field to an empty string (`""`).

simulation.view

Testbench view name. Usually `adexl`.

simulation.state

ADE-XL setup state name. When you run simulations from BAG, the simulation configuration will be saved to this setup state.

simulation.update_timeout_ms

If simulation takes a lone time, BAG will print out a message at this time interval (in milliseconds) so you can know if BAG is still running.

simulation.kwargs

pexpect keyword arguments dictionary used to start the simulation. When BAG server receive a simulation request, it will run Ocean in a subprocess using Python pexpect module. This entry allows you to control how pexpect starts the Ocean subprocess. Refer to pexpect documentation for more information.

job_options

A dictionary of job options for ADE-XL. This entry controls whether ADE-XL runs simulations remotely or locally, and how many jobs it launches for a simulation run. Refer to ADE-XL documentation for available options.

3.4.4 class

The subclass of :ref:

3.4.5 lib_defs

Location of the BAG design module libraries definition file.

The BAG libraries definition file is similar to the `cds.lib` file for Virtuoso, where it defines every design module library and its location. This file makes it easy to share design module libraries made by different designers.

Each line in the file contains two entries, separated by spaces. The first entry is the name of the design module library, and the second entry is the location of the design module library. Environment variables may be used in this file.

3.4.6 new_lib_path

Directory to put new generated design module libraries.

When you import a new schematic generator library, BAG will create a corresponding Python design module library and define this library in the library definition file (see *lib_defs*). This field tells BAG where new design module libraries should be created.

3.5 Technology Configuration File

Technology configuration file is written in YAML format. This document describes each setting. Technology configuration file may use environment variable to specify values of any entries.

3.5.1 class

The subclass of `bag.layout.core.TechInfo` for this process technology. If this entry is not defined, a default dummy `TechInfo` instance will be created for schematic-only design flow.

3.5.2 mos

This entry defines all MOS transistor settings.

mos.width_resolution

The transistor width minimum resolution, in meters or number of fins in finfet technology.

mos.length_resolution

The transistor length minimum resolution, in meters.

mos.mos_char_root

The default transistor characterization data directory.

3.5.3 layout

This entry defines all layout specific settings.

layout.em_temp

The temperature used to calculate electro-migration specs. The temperature should be specified in degrees Celsius.

3.6 Setting up New PDK

This section describes how to get BAG 2.0 to work with a new PDK.

1. Create a new technology configuration file for this PDK. See *Technology Configuration File* for a description of the technology configuration file format.
2. Create a new BAG configuration file for this PDK. You can simply copy an existing configuration, then change the fields listed in *Changing Process Technology*.
3. Create a new `BAG_prim` library for this PDK. The easiest way to do this is to copy an existing `BAG_prim` library, then change the underlying instances to be instances from the new PDK. You should use the **pPar** command in Virtuoso to pass CDF parameters from `BAG_prim` instances to PDK instances.
4. Change your cds.lib to refer to the new `BAG_prim` library.
5. To avoid everyone having their own python design modules for BAG primitive, you should generated a global design module library for BAG primitives, then ask every user to include this global library in their `bag_libs.def` file. To do so, setup a BAG workspace and execute the following commands:

   ```
   import bag
   prj = bag.BagProject()
   prj.import_design_library('BAG_prim')
   ```

   now copy the generate design library to a global location.

CHAPTER 4

Developer Guide

Nothing here yet...

CHAPTER 5

bag

5.1 bag package

5.1.1 Subpackages

bag.data package

Submodules

bag.data.core module

bag.data.dc module

bag.data.digital module

bag.data.lti module

bag.data.ltv module

bag.data.mos module

bag.data.plot module

Module contents

bag.design package

Submodules

5.1.2 Submodules

5.1.3 bag.core module

5.1.4 bag.virtuoso module

5.1.5 Module contents

CHAPTER 6

Indices and tables

- genindex
- modindex
- search