ACG Documentation

Release 0.1.0

Pavan Bhargava

May 03, 2020

CONTENTS:

Ayar Custom Generator(ACG) is a package that enables full custom layout creation without a grid. It allows a designer to describe layout generation script in the Python language and automate the layout process

CHAPTER

ONE

QUICK START GUIDE

This chapter contains the fastest guide to getting your first generated layout.

1.1 Installation

TODO: Step-by-step procedure to set up ACG

1. clone and pip install ACG
2. set up example technology files
3. run a quick wiring example
4. show how to incorporate with cadence

1.2 First Layout

TODO: Step-by-step procedure to run a basic generator with a wire and a via

1. Show where the example script is
2. Describe how to run the script
3. Place a picture of what the output should look like

1.3 Cadence Interface

TODO: Step-by-step procedure to set up the Cadence interface, and generate your first OA file

1. Describe required lines to be added to the cdsinit files
2. Show how to change the generator to make an OA vs. a GDS

CHAPTER

TWO

GETTING STARTED

This chapter contains a more detailed guide to understanding how to use the core ACG feature set

2.1 Design Flow

TODO: Explain high level architecture of ACG

1. All layout scripts are written as classes which subclass `AyarLayoutGenerator`
2. Your subclass of `AyarLayoutGenerator` defines how the layout should be drawn in terms of parameters that you define and technology constants
3. `AyarDesignManager` configures details about your technology, receives parameters from the user, and runs the generator to create an output layout.
4. Talk about spec files, and why they are useful

2.2 Anatomy of a Generator

TODO: Explain detailed information about the ALG class

1. `AyarLayoutGenerator` provides helper methods to place and manipulate shapes and hierarchically place other instances and run other generators. Consider it to be your palette of functions you can use to describe your layout procedure.
2. `get_params_info()` should be defined to set which parameters your generator expects.
3. `AyarLayoutGenerator` defines an abstractmethod `draw_layout()` which contains the layout drawing algorithm in terms of your desired parameters.

Show pictures of how a few of the key functions work

2.3 Creating Wires

TODO: Explain how to create and move rectangles around

1. Explain philosophy of relative alignment and stretching
2. Provide examples of how to use the alignment and stretching functions, explain rectangle handles and edges.
3. Talk about tracks.
4. Briefly explain via creation methodology.

2.4 Creating Hierarchical Cells

TODO: Explain how to create an inverter

1. Show how to run a generator within another generator
2. Explain the difference between masters and instances
3. Show how to place and align instances
4. Emphasize cacheing of masters
5. Example design of an inverter

CHAPTER

THREE

COMMON USE CASES

This chapter contains a set of examples that describe how to do several common design tasks.

3.1 Incorporating Hard IP with ACG Generators

TODO: Show how to make a buffer chain with custom designs

1. Describe how the cadence interface works
2. Show how to import a hand-generated cell as a master
3. Show an example of a cascading buffer chain

3.2 AMS Floorplanning and Assembly

TODO: Show a floorplanning design methodology.

1. Provide an example of quick placeholder layout arrangement
2. Describe a flow where designers incrementally update cells and how to adapt with floorplanning
3. Show power stripe creation

3.3 Autorouting

TODO: Show all the different autorouter types available

3.4 Transistor Drawing and Row Generation

1. Show an example of a transistor drawing generator with the example technology
2. Show how to use transistor row generators

CHAPTER

FOUR

ACG

4.1 ACG package

4.1.1 Submodules

4.1.2 ACG.AutoRouter module

4.1.3 ACG.AutoRouterExtension module

4.1.4 ACG.AyarDesignManager module

class ACG.AyarDesignManager.**AyarDesignManager**(*bprj*, *spec_file*, *gds_layermap=''*)
Bases: object

Class that oversees the creation of layouts, schematics, testbenches, and simulations. Overrides DesignMaster for more intuitive yaml file organization and handles the RoutingGrid in grid-free layouts

generate_layout(*layout_params_list=None*, *cell_name_list=None*)
Generates a batch of layouts with the layout package/class in the spec file with parameters set by layout_params_list and names them according to cell_name_list. Each dict in the layout_params_list creates a new layout

layout_params_list [:obj:'list' of :obj:'dict'] list of parameter dicts to be applied to the specified layout class

cell_name_list [:obj:'list' of :obj:'str'] list of names to be applied to each implementation of the layout class

generate_schematic(*sch_params_list=None*, *cell_name_list=None*)
Generates a batch of schematics specified by sch_params_list and names them according to cell_name_list. Each dict in the sch_params_list creates a new schematic

Parameters

- **sch_params_list** (*:obj:'list' of :obj:'dict'*) – parameter dicts to be applied to the specified layout class
- **cell_name_list** (*:obj:'list' of :obj:'str'*) – list of names to be applied to each implementation of the layout class

generate_tb(*tb_params_list=None*, *tb_name_list=None*)
Generates a batch of testbenches specified by tb_params_list and names them according to tb_name_list. Each dict in tb_params_list creates a new set of tb's

Parameters

- **tb_params_list** (*:obj:'list' of :obj:'dict'*) – list of parameter dicts to be applied to the testbench generator class
- **tb_name_list** (*:obj:'list' of :obj:'str'*) – list of names to be applied to each implementation of the tb class

import_schematic_library(*lib_name*)
Imports a Cadence library containing schematic templates for use in BAG, this must be called if changes to the schematic were made since the last run

Parameters **lib_name** (*str*) – string containing name of the library to be imported

load_sim_data()
Returns simulation data for all TBs in spec file

make_tdb(*layermap=''*)
Makes a new TemplateDB object. If no routing grid parameters are sent in, dummy parameters are used.

run_LVS(*cell_name_list=None*)
Runs LVS on a batch of cells contained within the implementation library

Parameters **cell_name_list** (*:obj:'list' of :obj:'str'*) – list of strings containing the names of the cells we should run LVS on

run_PEX(*cell_name_list*)
Runs PEX on a batch of cells contained within the implementation library

Parameters **cell_name_list** (*:obj:'list' of :obj:'str'*) – list of strings containing the names of the cells we should run PEX on

run_flow()
Override and call this method to specify your design procedure when subclassing

simulate()
Runs a batch of simulations on the generated TB's. All parameters for simulation are set within the spec file

4.1.5 ACG.AyarLayoutGenerator module

4.1.6 ACG.Label module

4.1.7 ACG.LayoutParse module

4.1.8 ACG.PrimitiveUtil module

Utility functions

ACG.PrimitiveUtil.**Md**(*direction*)
Get direction/projection matrix

Parameters **direction** (*str*) – direction/projection parameter. Possible values are 'left', 'right', 'top', 'bottom', 'omni', 'x', 'y'.

Returns directional matrix

Return type np.array([[int, int], [int, int]])

ACG.PrimitiveUtil.**Mt**(*transform*)
Get transform matrix

Parameters **transform** (*str*) – transform parameter. possible values are 'R0', 'MX', 'MY', 'MXY', and 'R180'

Returns transform matrix

Return type np.array([[int, int], [int, int]])

ACG.PrimitiveUtil.**Mtinv**(*transform*)
Get inverse of transform matrix

Parameters **transform** (*str*) – transform parameter. possible values are 'R0', 'MX', 'MY', 'MXY', and 'R180'

Returns inverse of transform matrix

Return type np.array([[int, int], [int, int]])

ACG.PrimitiveUtil.**format_float**(*value*, *res*)
Format float numbers for pretty printing

Parameters

- **value** (*float*) – number to be printed
- **res** (*float*) – resolution

Returns

Return type str

ACG.PrimitiveUtil.**locate_xy**(*xy0*, *xy1*, *location*)
Find a corresponding xy coordinate from location parameters

Parameters

- **xy0** (*np.array([float, float])*) – first coordinate
- **xy1** (*np.array([float, float])*) – second coordinate
- **location** (*str*) – direction/projection parameter. Possible values are 'lowerLeft', 'upperRight', 'centerCenter', ...

Returns resulting coordinate

Return type np.array([float, float])

4.1.9 ACG.Rectangle module

class ACG.Rectangle.**Rectangle**(*xy*, *layer*, *virtual=False*)
Bases: *ACG.VirtualObj.VirtualObj*

Creates a better rectangle object with stretch and align capabilities

align(*target_handle: str*, *ref_rect: Optional[*ACG.Rectangle.Rectangle*] = None*, *ref_handle: str = None*, *track=None*, *align_opt: Tuple[bool, bool] = True, True*, *offset: Union[Tuple[float, float],* ACG.XY.XY*] = 0, 0*) → *ACG.Rectangle.Rectangle*
Moves the rectangle to co-locate the target and ref handles

property center

copy(*virtual=False*, *layer=None*) → *ACG.Rectangle.Rectangle*

dict_compatability = ('handle0', 'handle1', 'xy0', 'xy1', 'layer')

export_locations()
This method should return a dict of relevant locations for the virtual obj

classmethod from_dict(*params: dict*) → *ACG.Rectangle.Rectangle*
Enable the creation of a rectangle from a dictionary of parameters

get_dim(*dim*)
Returns measurement of the dimension of the rectangle

get_enclosure(*rect:* ACG.Rectangle.Rectangle, *virtual: bool = True*) → *ACG.Rectangle.Rectangle*
Returns a rectangle that encloses all provided rectangles

get_highest_layer(*rect: Optional[Rectangle] = None*, *layer: Optional[str] = None*) → Tuple[str, str]
Returns the highest layer used by provided rectangles

get_midpoint(*handle: str*, *coord: Union[Tuple[float, float],* ACG.XY.XY*]*) → Union[Tuple[float, float], *ACG.XY.XY*]
Gets the midpoint between a location on this rectangle and another coordinate

get_overlap(*rect:* ACG.Rectangle.Rectangle, *virtual: bool = True*) → *ACG.Rectangle.Rectangle*
Returns a rectangle corresponding to the overlapped region between two rectangles

property height

property layer

property ll

property lpp

static overlap(*A*, *B*)
Returns whether or not two rectangles overlap in both dimensions

scale(*size*, *dim=None*) → *ACG.Rectangle.Rectangle*
Additvely resizes the rectangle by the provided size

set_dim(*dim: str*, *size: float*) → *ACG.Rectangle.Rectangle*
Sets either the width or height of the rect to desired value. Maintains center location of rect

shift_origin(*origin=0, 0*, *orient='R0'*, *virtual=True*) → *ACG.Rectangle.Rectangle*
Takes xy coordinates and rotation, returns a virtual Rect2 that is re-referenced to the new origin Assumes that the origin of the rectangle is (0, 0)

stretch(*target_handle: str*, *ref_rect=None*, *ref_handle: str = None*, *track=None*, *stretch_opt=True, True*, *offset=0, 0*) → *ACG.Rectangle.Rectangle*
Stretches rectangle to co-locate the target and ref handles. If ref handles are not provided, stretch by given offset

to_bbox() → bag.layout.util.BBox

update_dict()
Updates the location dictionary based on the current ll and ur coordinates

property ur

property width

property xy

4.1.10 ACG.Track module

class ACG.Track.**Track**(*dim*, *spacing*, *origin=0*)
Bases: object

A class for creating consistently spaced reference lines

align(*ref_rect*, *ref_handle*, *num=0*, *offset=0*)
Aligns the provided track number to handle of reference rectangle

property dim

get_track(*num*) → *ACG.XY.XY*
Returns [x, y] coordinates of desired track #

property origin

property spacing

stretch(*track_num*, *ref_rect*, *ref_handle*, *offset=0*)
Stretches the track spacing to co-locate track and handle

class ACG.Track.**TrackManager**
Bases: object

A class that enables users to create tracks and use them as references for routing

add_track(*name*, *dim*, *spacing*, *origin=0*)
Adds a track to the database.

Parameters

- **name** – name to associate with the added track
- **dim** – 'x' or 'y' for the desired routing direction
- **spacing** – space between tracks
- **origin** – coordinate to place the zero track

classmethod from_routing_grid(*grid: RoutingGrid*)
Generates a track manager object from the current grid

Parameters **grid** – RoutingGrid object used as reference to build all of the desired tracks

Returns Generated TrackManager object from the provided RoutingGrid

Return type *TrackManager*

4.1.11 ACG.Via module

4.1.12 ACG.VirtualInst module

class ACG.VirtualInst.**VirtualInst**(*master*, *origin=0, 0*, *orient='R0'*, *inst_name=None*)
Bases: *ACG.VirtualObj.VirtualObj*

A class to enable movement/access of low level instances without directly accessing the master class

align(*target_handle: str*, *target_rect:* ACG.VirtualObj.VirtualObj *= None*, *ref_rect:* ACG.VirtualObj.VirtualObj *= None*, *ref_handle: str = None*, *align_opt: Tuple[bool, bool] = True, True*, *offset: Union[Tuple[Union[float, int], Union[float, int]],* ACG.XY.XY*] = 0, 0*) → *ACG.VirtualInst.VirtualInst*
Moves the instance to co-locate target rect handle and reference rect handle

edges = ('l', 'b', 'r', 't')

export_locations() → dict
Recursively shift all of the elements in the location dictionary

move(*origin=None*, *orient=None*) → *ACG.VirtualInst.VirtualInst*
Set the origin and orientation to new values

property orient

property origin

shift_origin(*origin=None*, *orient=None*) → *ACG.VirtualInst.VirtualInst*
Moves the instance origin to the provided coordinates and performs transformation

valid_orientation = ('R0', 'MX', 'MY', 'R180')

vertices = ('ll', 'lr', 'ur', 'ul', 'c', 'cl', 'cb', 'cr', 'ct')

4.1.13 ACG.VirtualObj module

class ACG.VirtualObj.**VirtualObj**
Bases: object

Abstract class for creation of primitive objects

export_locations()
This method should return a dict of relevant locations for the virtual obj

abstract shift_origin(*origin=0, 0*, *orient='R0'*)
This method should shift the coordinates of relevant locations according to provided origin/transformation, and return a new shifted object. This is important to allow for deep manipulation in the hierarchy

4.1.14 ACG.XY module

class ACG.XY.**XY**(*xy*, *res=0.001*)
Bases: *ACG.VirtualObj.VirtualObj*

Primitive class to describe a single coordinate on xy plane and various associated utility functions Keeps all coordinates on the grid

export_locations()
For now just returns a dict of the coordinates

shift_origin(*origin=0, 0*, *orient='R0'*)
This method should shift the coordinates of relevant locations according to provided origin/transformation, and return a new shifted object. This is important to allow for deep manipulation in the hierarchy

property x

property xy

property y

4.1.15 ACG.tech module

When imported, this module will extract tech information for easy access from a module specified by the 'ACG_TECH' environment variable

4.1.16 Module contents

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SIX

INTRODUCTION

Documentation for available functions and classes can be found at modindex

Arbitrary Cell Generator is a plugin to Berkeley Analog Generator 2.0 (BAG) that enables parametrized grid-free layout creation. The main goal of ACG is to make it easy to create highly customized layout generators that may not be possible in BAG due to its requirement to make layouts technology agnostic. In addition, we make it easy to combine legacy hand-drawn layouts and IP together with new layout generators.

NOTE: ACG is currently in development, and is being slowly cleaned up for open-source consumption, use at your own risk!

PYTHON MODULE INDEX

a

INDEX

M

O

R

S

T

U

V

W

X

Y