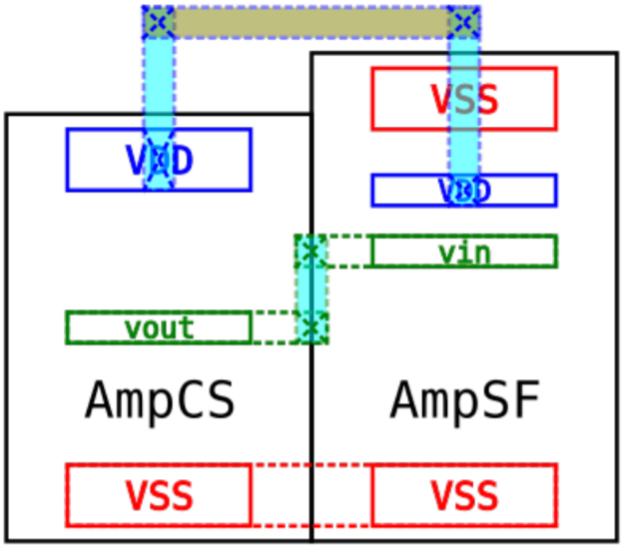
Module 5: Hierarchical Generators

This module covers writing layout/schematic generators that instantiate other generators. We will write a two-stage amplifier generator, which instatiates the common-source amplifier followed by the source-follower amplifier.

AmpChain Layout Example

First, we will write a layout generator for the two-stage amplifier. The layout floorplan is drawn for you below:



This floorplan abuts the `AmpCS` instance next to `AmpSF` instance, the `VSS` ports are simply shorted together, and the top `VSS` port of `AmpSF` is ignored (they are connected together internally by dummy connections). The intermediate node of the two-stage amplifier is connected using a vertical routing track in the middle of the two amplifier blocks. `VDD` ports are connected to the top-most M6 horizontal track, and other ports are simply exported in-place.

The layout generator is reproduced below, with some parts missing (which you will fill out later). We will walk through the important sections of the code.

```
class AmpChain(TemplateBase):
    def __init__(self, temp_db, lib_name, params, used_names, **kwargs
):
        TemplateBase.__init__(self, temp_db, lib_name, params, used_na
mes, **kwargs)
        self._sch_params = None

    @property
    def sch_params(self):
        return self._sch_params

    @classmethod
    def get_params_info(cls):
        return dict(
            cs_params='common source amplifier parameters.',
            sf_params='source follower parameters.',
            show_pins='True to draw pin geometries.',
        )

    def draw_layout(self):
        """Draw the layout of a transistor for characterization.
        """
```

AmpChain Constructor

```
class AmpChain(TemplateBase):
    def __init__(self, temp_db, lib_name, params, used_names, **kwargs
):
        TemplateBase.__init__(self, temp_db, lib_name, params, used_na
mes, **kwargs)
        self._sch_params = None

    @property
    def sch_params(self):
        return self._sch_params

    @classmethod
    def get_params_info(cls):
        return dict(
            cs_params='common source amplifier parameters.',
            sf_params='source follower parameters.',
            show_pins='True to draw pin geometries.',
        )
```

First, notice that instead of subclassing `AnalogBase`, the `AmpChain` class subclasses `TemplateBase`. This is because we are not trying to draw transistor rows inside this layout generator; we just want to place and route multiple layout instances together. `TemplateBase` is the base class for all layout generators and it provides most placement and routing methods you need.

Next, notice that the parameters for `AmpChain` are simply parameter dictionaries for the two sub-generators. The ability to use complex data structures as generator parameters solves the parameter explosion problem when writing generators with many levels of hierarchy.

Creating Layout Master

```
# create layout masters for subcells we will add later
cs_master = self.new_template(params=cs_params, temp_cls=AmpCS)
# TODO: create sf_master.  Use AmpSFSoln class
sf_master = None
```

Here, the `new_template()` function creates a new layout master, `cs_master`, which represents a generated layout cellview from the `AmpCS` layout generator. We can later instances of this master in the current layout, which are references to the generated `AmpCS` layout cellview, perhaps shifted and rotated. The main take away is that the `new_template()` function does not add any layout geometries to the current layout, but rather create a separate layout cellview which we may use later.

Creating Layout Instance

```
# add subcell instances
cs_inst = self.add_instance(cs_master, 'XCS')
# add source follower to the right of common source
x0 = cs_inst.bound_box.right_unit
sf_inst = self.add_instance(sf_master, 'XSF', loc=(x0, 0), unit_mode=T
rue)
```

The `add_instance()` method adds an instance of the given layout master to the current cellview. By default, if no location or orientation is given, it puts the instance at the origin with no rotation. the `bound_box` attribute can then be used on the instance to get the bounding box of the instance. Here, the bounding box is used to determine the X coordinate of the source-follower.

Get Instance Ports

```
# get subcell ports as WireArrays so we can connect them
vmid0 = cs_inst.get_all_port_pins('vout')[0]
vmid1 = sf_inst.get_all_port_pins('vin')[0]
vdd0 = cs_inst.get_all_port_pins('VDD')[0]
vdd1 = sf_inst.get_all_port_pins('VDD')[0]
```

after adding an instance, the `get_all_port_pins()` function can be used to obtain a list of all pins as `WireArray` objects with the given name. In this case, we know that there's exactly one pin, so we use Python list indexing to obtain first element of the list.

Routing Grid Object

```
# get vertical VDD TrackIDs
vdd0_tid = TrackID(vm_layer, self.grid.coord_to_nearest_track(vm_layer
, vdd0.middle))
vdd1_tid = TrackID(vm_layer, self.grid.coord_to_nearest_track(vm_layer
, vdd1.middle))
```

the `self.grid` attribute of `TemplateBase` is a `RoutingGrid` objects, which provides many useful functions related to the routing grid. In this particular scenario, `coord_to_nearest_track()` is used to determine the vertical track index closest to the center of the `VDD` ports. These vertical tracks will be used later to connect the `VDD` ports together.

Re-export Pins on Instances

```
    # re-export pins on subcells.
    self.reexport(cs_inst.get_port('vin'), show=show_pins)
    self.reexport(cs_inst.get_port('vbias'), net_name='vb1', show=show_pin
    s)
    # TODO: reexport vout and vbias of source follower
    # TODO: vbias should be renamed to vb2
```

`TemplateBase` also provides a `reexport()` function, which is a convenience function to re-export an instance port in-place. The `net_name` optional parameter can be used to change the port name. In this example, the `vbias` port of common-source amplifier is renamed to `vb1`.

Layout Exercises

Now you should know everything you need to finish the two-stage amplifier layout generator. Fill in the missing pieces to do the following:

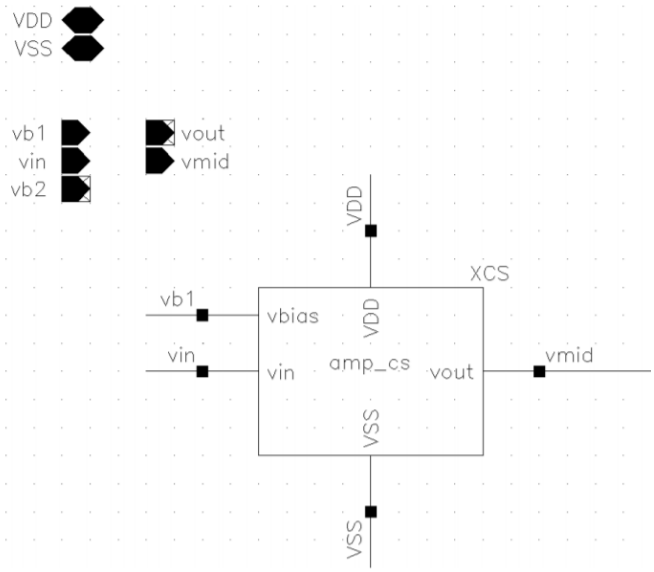
1. Create layout master for `AmpSF` using the `AmpSFSoln` class.
2. Using `RoutingGrid`, determine the vertical track index in the middle of the two amplifier blocks, and connect `vmid` wires together using this track.
    - Hint: variable `x0` is the X coordinate of the boundary between the two blocks.
3. Re-export `vout` and `vbias` of the source-follower. Rename `vbias` to `vb2`.

Once you're done, evaluate the cell below, which will generate the layout and run LVS. If everything is done correctly, a layout should be generated inthe `DEMO_AMP_CHAIN` library, and LVS should pass.

In [1]:
```
from bag.layout.routing import TrackID
from bag.layout.template import TemplateBase

from xbase_demo.demo_layout.core import AmpCS, AmpSFSoln


class AmpChain(TemplateBase):
    def __init__(self, temp_db, lib_name, params, used_names, **kwargs):
        TemplateBase.__init__(self, temp_db, lib_name, params, used_names, **
        self._sch_params = None

    @property
    def sch_params(self):
        return self._sch_params

    @classmethod
    def get_params_info(cls):
        return dict(
            cs_params='common source amplifier parameters.',
            sf_params='source follower parameters.',
            show_pins='True to draw pin geometries.',
        )

    def draw_layout(self):
        """Draw the layout of a transistor for characterization.
        """

        # make copies of given dictionaries to avoid modifying external data.
        cs_params = self.params['cs_params'].copy()
        sf_params = self.params['sf_params'].copy()
        show_pins = self.params['show_pins']

        # disable pins in subcells
        cs_params['show_pins'] = False
        sf_params['show_pins'] = False

        # create layout masters for subcells we will add later
        cs_master = self.new_template(params=cs_params, temp_cls=AmpCS)
        # TODO: create sf_master.  Use AmpSFSoln class
        sf_master = None

        if sf_master is None:
            return

        # add subcell instances
        cs_inst = self.add_instance(cs_master, 'XCS')
        # add source follower to the right of common source
        x0 = cs_inst.bound_box.right_unit
        sf_inst = self.add_instance(sf_master, 'XSF', loc=(x0, 0), unit_mode=

        # get VSS wires from AmpCS/AmpSF
        cs_vss_warr = cs_inst.get_all_port_pins('VSS')[0]
        sf_vss_warrs = sf_inst.get_all_port_pins('VSS')
        # only connect bottom VSS wire of source follower
        if len(sf_vss_warrs) < 2 or sf_vss_warrs[0].track_id.base_index < sf_
            sf_vss_warr = sf_vss_warrs[0]
        else:
            sf_vss_warr = sf_vss_warrs[1]

        # connect VSS of the two blocks together
        vss = self.connect_wires([cs_vss_warr, sf_vss_warr])[0]

        # get layer IDs from VSS wire
```

AmpChain Schematic Template

Now let's move on to schematic generator. As before, we need to create the schematic template first. A half-complete schematic template is provided for you in library `demo_templates`, cell `amp_chain`, shown below:



The schematic template for a hierarchical generator is very simple; you simply need to instantiate the schematic templates of the sub-blocks (***Not the generated schematic!***). For the exercise, instantiate the `amp_sf` schematic template from the `demo_templates` library, named it `XSF`, connect it, then evaluate the following cell to import the `amp_chain` netlist to Python.

In [11]:
```
import bag

# obtain BagProject instance
local_dict = locals()
if 'bprj' in local_dict:
    print('using existing BagProject')
    bprj = local_dict['bprj']
else:
    print('creating BagProject')
    bprj = bag.BagProject()

print('importing netlist from virtuoso')
bprj.import_design_library('demo_templates')
print('netlist import done')
```

```
using existing BagProject
importing netlist from virtuoso
netlist import done
```

AmpChain Schematic Generator

With schematic template done, you are ready to write the schematic generator. It is also very simple, you just need to call the `design()` method, which you implemented previously, on each instance in the schematic. Complete the following schematic generator, then evaluate the cell to push it through the design flow.

In [2]:
```
%matplotlib inline

import os

from bag.design import Module


# noinspection PyPep8Naming
class demo_templates__amp_chain(Module):
    """Module for library demo_templates cell amp_chain.

    Fill in high level description here.
    """

    # hard coded netlist flie path to get jupyter notebook working.
    yaml_file = os.path.join(os.environ['BAG_WORK_DIR'], 'BAG_XBase_demo',
                             'BagModules', 'demo_templates', 'netlist_info',

    def __init__(self, bag_config, parent=None, prj=None, **kwargs):
        Module.__init__(self, bag_config, self.yaml_file, parent=parent, prj=

    @classmethod
    def get_params_info(cls):
        # type: () -> Dict[str, str]
        """Returns a dictionary from parameter names to descriptions.

        Returns
        -------
        param_info : Optional[Dict[str, str]]
            dictionary from parameter names to descriptions.
        """
        return dict(
            cs_params='common-source amplifier parameters dictionary.',
            sf_params='source-follwer amplifier parameters dictionary.',
        )

    def design(self, cs_params=None, sf_params=None):
        self.instances['XCS'].design(**cs_params)
        # TODO: design XSF


import os

# import bag package
import bag
from bag.io import read_yaml

# import BAG demo Python modules
import xbase_demo.core as demo_core
from xbase_demo.demo_layout.core import AmpChainSoln

# load circuit specifications from file
spec_fname = os.path.join(os.environ['BAG_WORK_DIR'], 'specs_demo/demo.yaml')
top_specs = read_yaml(spec_fname)

# obtain BagProject instance
local_dict = locals()
if 'bprj' in local_dict:
    print('using existing BagProject')
    bprj = local_dict['bprj']
else:
    print('creating BagProject')
    bprj = bag.BagProject()
```