Module 2: XBase Routing API

In this module, you will learn the basics about the routing grid system in XBase. We will go over how tracks are defined, how to create wires, vias and pins, and how to define the size of a layout cell.

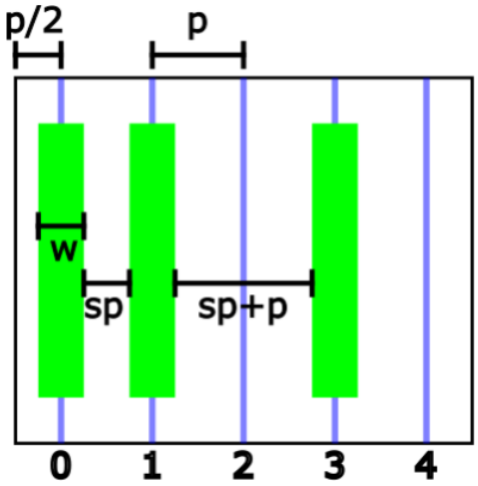
XBase Routing Grid

```
routing_grid:
    layers: [4, 5, 6, 7]
    spaces: [0.06, 0.1, 0.12, 0.2]
    widths: [0.06, 0.1, 0.12, 0.2]
    bot_dir: 'x'
```

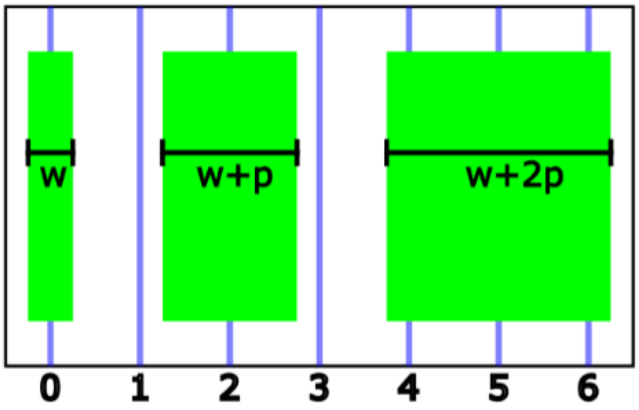
In XBase, all wires and vias have to be drawn on the routing grid, which is usually defined in a specification file, as shown above. On each layer, all wires must travel in the same direction (horizontal or vertical), and wire direction alternates between each layers. The routing grid usually starts on an intermediate layer (metal 4 in the above example), and lower layers are reserved for device primitives routing. As seen above, different layers can define different wire pitch, with the wire pitch generally increasing as you move up the metal stack.

All layout cell dimensions in XBase must also be quantized to the routing grid, meaning that a layout cell must contain integer number of tracks on all metal layers it uses. Because of the difference in wire pitch, a layout cell that use more layers generally have coarser quantization compared with a layout cell that use fewer layers.

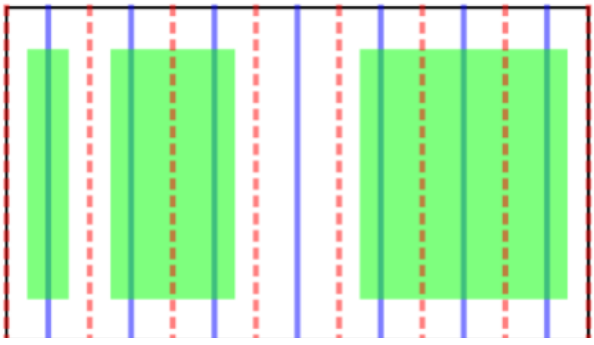
XBase Routing Tracks



The figure above shows some wires drawn in XBase. Track pitch is the sum of unit width and space, and track number 0 is defined as the wire that's half-pitch away from left or bottom boundary. From the figure, you can see spacing between wires follows the formula $S = sp + N \cdot p$, where N is the number of tracks in between.



XBase also supports drawing thicker wires by using multiple adjacent tracks. Wire width follows the formula $W = w + (N - 1) \cdot p$, where N is the number of tracks a wire uses. One issue with this scheme is that even width wires wastes more space compared to odd width wires. For example, in the above figure, although tracks 1 and 3 are empty, no wire can be drawn there because it will violate minimum spacing rule to the wire centered on track 2. As a result, the wire on track 2 takes up 3 tracks although it is only 2 tracks wide.



TrackID and WireArray

```
class TrackID(object):
    def __init__(self, layer_id, track_idx, width=1, num=1, pitch=0.0)
:
        #type: (int, Union[float, int], int, int, Union[float, int]) -
> None

class WireArray(object):
    def __init__(self, track_id, lower, upper):
        #type: (TrackID, float, float) -> None
```

Routing track locations are represented by the `TrackID` Python object. It has built-in support for drawing a multi-wire bus by specifying the optional `num` and `pitch` parameters, which defines the number of wires in the bus and the number of track pitches between adjacent wires. The `layer_id` parameter specifies the routing layer ID, the `track_idx` parameter specifies the track index of the left-most/bottom-most wire, and `width` specifies the number of tracks a single wire uses.

Physical wires in XBase are represented by the `WireArray` Python object. It contains a `TrackID` object describes the location of the wires, and `lower` and `upper` attributes describes the starting and ending coordinate of those wires along the track. For example, a horizontal wire starting at $x = 0.5$ um and ending at $x = 3.0$ um will have `lower = 0.5` , and `upper = 3.0` .

One last note is that layout pins can only be added on `WireArray` objects. This guarantees that pins of a layout cell will always be on the routing grid.

BAG Layout Generation Code

```
def gen_layout(prj, specs, dsn_name, demo_class):
    # get information from specs
    dsn_specs = specs[dsn_name]
    impl_lib = dsn_specs['impl_lib']
    layout_params = dsn_specs['layout_params']
    gen_cell = dsn_specs['gen_cell']

    # create layout template database
    tdb = make_tdb(prj, specs, impl_lib)
    # compute layout
    print('computing layout')
    # template = tdb.new_template(params=layout_params, temp_cls=temp_
cls)
    template = tdb.new_template(params=layout_params, temp_cls=demo_cl
ass)

    # create layout in OA database
    print('creating layout')
    tdb.batch_layout(prj, [template], [gen_cell])
    # return corresponding schematic parameters
    print('layout done')
    return template.sch_params
```

The above code snippet (taking from `xbase_demo.core` module) shows how layout is generated. First, user create a layout database object, which keeps track of layout hierarchy. Then, user uses the layout database object to create new layout instances given layout generator class and parameters. Finally, layout database uses `BagProject` instance to create the generated layouts in Virtuoso. The generated layout will also contain the corresponding schematic parameters, which can be passed to schematic generator later.

BAG TemplateDB Creation Code

```
def make_tdb(prj, specs, impl_lib):
    grid_specs = specs['routing_grid']
    layers = grid_specs['layers']
    spaces = grid_specs['spaces']
    widths = grid_specs['widths']
    bot_dir = grid_specs['bot_dir']

    # create RoutingGrid object
    routing_grid = RoutingGrid(prj.tech_info, layers, spaces, widths,
bot_dir)
    # create layout template database
    tdb = TemplateDB('template_libs.def', routing_grid, impl_lib, use_
cybagoa=True)
    return tdb
```

For reference, the above code snippet shows how the layout database object is created. A `RoutingGrid` object is created from routing grid parameters specified in the specification file, which is then used to construct the `TemplateDB` layout database object.

Routing Example

The code box below defines a `RoutingDemo` layout generator class, which is a simply layout containing only wires, vias, and pins. All layout creation happens in the `draw_layout()` function, the functions/attributes of interests are:

- `add_wires()` : Create one or more physical wires, with the given options.
- `connect_to_tracks()` : Connect two `WireArray` s on adjacent layers by extending them to their intersection and adding vias.
- `connnect_wires()` : Connect multiple `WireArrays` on the same layer together.
- `add_pin()` : Add a pin object on top of a `WireArray` object.
- `self.size` : A 3-tuple describing the size of this layout cell.
- `self.bound_box` : A `BBox` object representing the bounding box of this layout cell, computed from `self.size` .

To see the layout in action, evaluate the code box below by selecting the cell and pressing Ctrl+Enter. A `DEMO_ROUTING` library will be created in Virtuoso with a single `ROUTING_DEMO` layout cell in it. Feel free to play around with the numbers and re-evaluating the cell, and the layout in Virtuoso should update.

Exercise 1: There are currently 3 wires labeled "pin3". Change that to 4 wires by adding an extra wire with the same pitch on the right.

Exercise 2: Connect all wires labeled "pin3" to the wire labeled "pin1". Hint: use `connect_to_tracks()` and `connect_wires()` .

In [9]:
```
from bag.layout.routing import TrackID
from bag.layout.template import TemplateBase


class RoutingDemo(TemplateBase):
    def __init__(self, temp_db, lib_name, params, used_names, **kwargs):
        super(RoutingDemo, self).__init__(temp_db, lib_name, params, used_nam

    @classmethod
    def get_params_info(cls):
        return {}

    def draw_layout(self):
        # Metal 4 is horizontal, Metal 5 is vertical
        hm_layer = 4
        vm_layer = 5

        # add a horizontal wire on track 0, from X=0.1 to X=0.3
        warr1 = self.add_wires(hm_layer, 0, 0.1, 0.3)
        # print WireArray object
        print(warr1)
        # print lower, middle, and upper coordinate of wire.
        print(warr1.lower, warr1.middle, warr1.upper)
        # print TrackID object associated with WireArray
        print(warr1.track_id)

        # add a horizontal wire on track 1, from X=0.1 to X=0.3,
        # coordinates specified in resolution units
        warr2 = self.add_wires(hm_layer, 1, 100, 300, unit_mode=True)

        # add another wire on track 1, from X=0.35 to X=0.45
        warr2_ext = self.add_wires(hm_layer, 1, 350, 450, unit_mode=True)
        # connect wires on the same track, in this case warr2 and warr2_ext
        self.connect_wires([warr2, warr2_ext])

        # add a horizontal wire on track 2.5, from X=0.2 to X=0.4
        self.add_wires(hm_layer, 2.5, 200, 400, unit_mode=True)
        # add a horizontal wire on track 4, from X=0.2 to X=0.4, with 2 track
        warr3 = self.add_wires(hm_layer, 4, 200, 400, width=2, unit_mode=True

        # add 3 parallel vertical wires starting on track 6 and use every oth
        warr4 = self.add_wires(vm_layer, 6, 100, 400, num=3, pitch=2, unit_mo
        print(warr4)

        # create a TrackID object representing a vertical track
        tid = TrackID(vm_layer, 3, width=2, num=1, pitch=0)
        # connect horizontal wires to the vertical track
        warr5 = self.connect_to_tracks([warr1, warr3], tid)
        print(warr5)

        # add a pin on a WireArray
        self.add_pin('pin1', warr1)
        # add a pin, but make label different than net name.  Useful for LVS
        self.add_pin('pin2', warr2, label='pin2:')
        # add_pin also works for WireArray representing multiple wires
        self.add_pin('pin3', warr4)
        # add a pin (so it is visible in BAG), but do not create the actual l
        # in OA.  This is useful for hiding pins on lower levels of hierarchy
        self.add_pin('pin4', warr3, show=False)

        # set the size of this template
        top_layer = vm_layer
        num_h_tracks = 6
```