GDS MILL

USER MANUAL

Version 1.0

AUTHOR: Michael Wieckowski
CONTACT: wieckows@umich.edu

Introduction

What is GDS Mill?

While GUI interfaces are typically well suited to designing VLSI layouts, there do exist many tasks where a scripted approach is the most efficient. A few common examples are the tiling of memory cells in the compilation of a cache array, the filling of a top level layout to meet density requirements, and the striping of a top level power grid. In these instances, one often wants to include parametrization to enable painless engineering changes during the design cycle. To do this, scripting large portions of the layout simply makes the most sense.

GDS Mill is a framework for scripting VLSI layouts in Python.

GDS Mill was designed from the ground up to be user friendly, highly extensible, and widely applicable without sacrificing ability or efficiency. In short, I want you to use it, and use it painlessly! GDS Mill is based on the industry standard binary GDSII file format as its input/output and integrates nicely with the Cadence design suite. Using GDS Mill, you can stream GDS layouts to and from Cadence, create layouts from scratch, or modify existing layouts, all with only a handful of code in Python. Why Python?

1. Available on almost every platform including Windows and OS X. Free!
2. Quick learning curve
3. Easy to read
4. Powerful - dynamic types, object oriented
5. Feature rich libraries, free IDE's, and much more!

GDS Mill Features

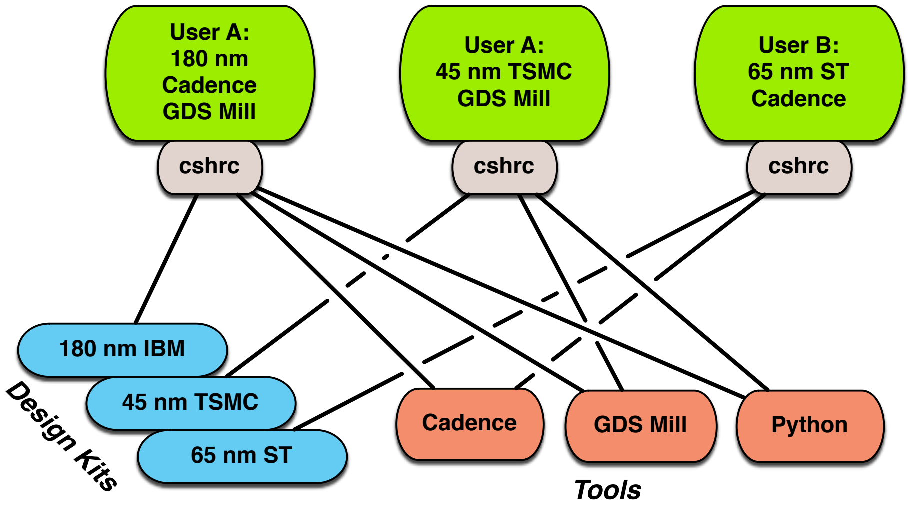
GDS Mill's first release (1.0) contains all the necessary tools to do layout scripting for most applications:

1. Read and write binary GDSII files
2. Stream to and from Cadence using PIPO
3. Create basic layout shapes including boxes, paths, and text
4. Instantiate one layout within another to create hierarchical designs
5. Perform basic area filling operations

GDS Mill is being developed rapidly and new features are being added daily. In addition, I welcome any feedback and feature requests from the VLSI community, so don't hesitate to ask if you think function X is a must have!

Setup

Process technology independence and user independence is a MUST for any successful VLSI project. To that end, I recommend setting up your design environment as shown in the following figure. In essence, all design kits and tools are installed in isolated locations to be shared among all users. Each user maintains separate directories for each of their projects, where each directory has a shell script to link the required tools for that project only. The following setup instructions for GDS Mill are based on this model.



Installing Python

Since GDS Mill is a tool for layout scripting in Python, it requires a recent version of Python to be installed (2.4 or higher). Most linux distributions will have a version of Python installed for you to use, but the version may not be recent enough. Luckily, you can install a local copy of Python on top of any existing installations if the machine you want to run on doesn't meet the requirements. I recommend installing a copy of Python 2.6 locally as follows:

1. Download Python via SVN and decompress it as follows:
   1.1. cd /some/tempPath/
   1.2. wget http://www.python.org/ftp/python/2.6.1/Python-2.6.1.tgz
   1.3. tar xvzf Python-2.6.1.tgz
2. Install Python into your home directory (or a project directory):
   2.1. cd /some/tempPath/Python-2.6.1
   2.2. ./configure –prefix /some/localPath/forPython/toReside
   2.3. make

    2.4. make install

Installing GDS Mill

These instructions will install GDS Mill into a central location.

1. Download the GDS Mill zip from http://www.vlsitools.com

2. mkdir /design/common/GDSMill

3. mv /download/location/GdsMill_1.0.zip /design/common/GDSMill

4. cd /design/common/GDSMill

5. unzip GdsMill_1.0.zip

6. rm GdsMill_1.0.zip

The remainder of the "installation" involves copying and modifying the shell script in /design/common/GDSMill/exampleUserDir. This is covered in the section "Setup a directory to work from" below.

Installing the NCSU CDK

All the examples in this manual are based on the NCSU Cadence Design Kit. Since it is freely available, it is a good platform to introducing GDS Mill while avoiding any IP issues. GDS Mill will work with ANY technology. The following steps were used to install the NCSU CDK on my system (yours might be different, so read carefully.)

1. Register and download the CDK from:
   http://www.eda.ncsu.edu/wiki/NCSU_CDK_download

2. I consider it good practice to keep the CDK itself in a central location that is referenced by each user's working directory. Untar / Unzip the downloaded CDK into your central location. For my case:

    2.1. cp ncsu-cdk-1.5.1.tar.gz /design/common

    2.2. cd /design/common

    2.3. tar -xvvzf ncsu-cdk-1.5.1.tar.gz

Setup a directory to work from

Your working directory will be the location where you run GDS Mill, Cadence, etc. In this manual, we will setup the working directory to use the NCSU CDK installed above.

1. Create a directory specific to the process node you want to use. In my case, this will be AMI Semiconductor 0.6 micron process.
   1.1. mkdir ~/design/600nmAmi
2. Copy the CDS initialization and library files from the CDK common directory into your working directory.
   2.1. cp /design/common/ncsu-cdk-1.5.1/.cdsinit ~/design/600nmAmi/
   2.2. cp /design/common/ncsu-cdk-1.5.1/cdssetup/cds.lib ~/design/600nmAmi/
3. The files we copied in step 2 rely on some environment variable to work properly. Since these variables are specific to the NCSU CDK, we will NOT put them in our global shell initialization file. Instead, make a local file that you will source every time you want to use the design kit. For my case, using CSHELL:
   3.1. cd ~/design/600nmAmi
   3.2. for vi, use vi 600nmAmi.cshrc. Alternately, use your favorite editor.
   3.3. Enter the following lines into the file, changing the paths where appropriate:

```
setenv SYSTEM_CDS_LIB_DIR /tools/ic-5.141_usr5/tools/dfII/samples
setenv CDSHOME /tools/ic-5.141_usr5
setenv CDK_DIR /design/common/ncsu-cdk-1.5.1
setenv CDS_MMSIM_DIR /tools/mmsim-7.0
setenv CDS_INST_DIR /tools/ic-5.141_usr5
```

4. Now we need to copy a few files for GDS Mill. Take a look in the /design/common/GDSMill/exampleUserDir. This directory contains what a typical work directory might look like to GDSMill. For this setup, let's just copy it all into our work directory.
   4.1. cp -r /design/common/GDSMill/exampleUserDir/* ~/design/600nmAmi

4.2. cd ~/design/600nmAmi

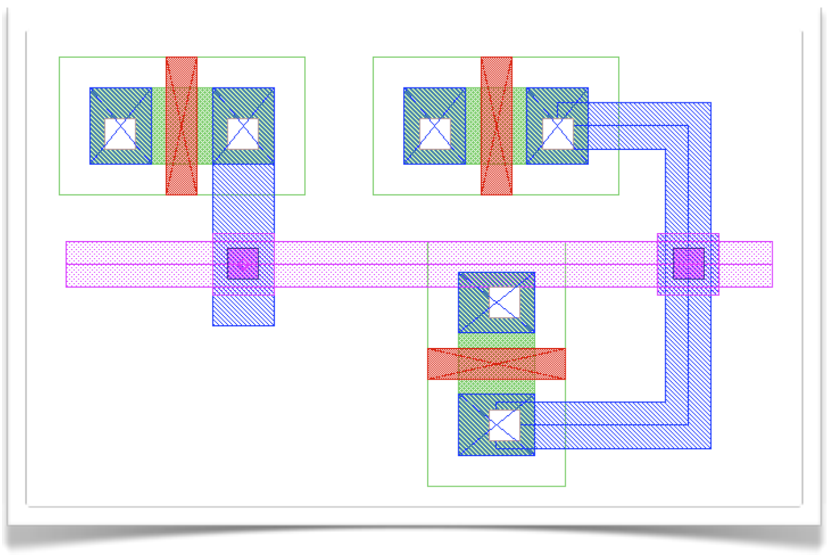
4.3. Edit the gdsMill.cshrc file according to the instructions inside. The most important variables to set are PATH and PYTHONPATH. Without these, you will likely have the wrong version of Python running and your scripts will not know where to find GDS Mill.

Quick Start Example

Create A Layout In Cadence

This section can be skipped if you want to just use a GDS file that already exists. A sample GDS file is included in the *exampleUserDir* directory.

1. Start Cadence using the NCSU AMI 0.6 CDK
    1.1. cd ~/design/600nmAmi
    1.2. source 600nmAmi.cshrc
    1.3. icfb &
2. Create a new library to work in
    2.1. CIW->Tools->Library Manager
    2.2. File->New->Library
    2.3. Enter "gdsMillTest" as the name
    2.4. Choose "Attach To An Existing Techfile"
    2.5. Select "NCSU_TechLib_ami06" as the technology library
3. Create a new layout to process using GDS Mill (still using library manager)
    3.1. File->New->Cellview
    3.2. Enter "testLayoutA" and the Cell Name and choose "Virtuoso" as the Tool
    3.3. Double click on the newly created layout file in the library manager to open Virtuoso
4. Create some shapes in the layout. An example is shown below where three NFET instances are placed, a Metal1 box is drawn, paths are added in Metal1 and Metal2, and a contact is added.



5. Save and close the layout window.

The GDS Mill Quick Start Script

The quick start script is located in the exampleUserDir directory. It gives a step by step example of using GDS Mill to export a layout from Cadence, modify it, stream the modified layout back into Cadence, and then output a PDF file of the modified layout.

To run the script:

1. cd ~/design/600nmAmi

2. edit quickStart.py so that the paths are real. For example, replace "~/design/600nmAmi" with an actual path, such as "/users/me/home/design/600nmAmi"

3. source gdsMill.cshrc

4. python quickStart.py

Let's go over the script piece by piece to learn the basics of GDS Mill. Lines 1 and 2 of the script simply import the gdsMill module into the Python environment.

```
1.  #!/usr/bin/env python
2.  import gdsMill
3.
```

Lines 4 through 13 create a GdsStreamer object called "streamer". The streamFromCadence method is then called with several parameters to export the layout that we made earlier into a

GDSII file. The resulting file will go into the gdsFiles directory of the exampleUserDir directory and will be called “testLayoutA.gds”.

```
4.  #creater a streamer object to interact with the Cadence libraries
5.  streamer = gdsMill.GdsStreamer()
6.
7.  #use the streamer to take a Cadence layout, and convert it to GDSII for us to work with
8.  #the GDS will be named testLayoutA.gds
9.  streamer.streamFromCadence(cadenceLibraryContainerPath = "~/design/600nmAmi",
10.                            libraryName = "gdsMillTest",
11.                            cellName = "testLayoutA",
12.                            outputPath = "./gdsFiles")
13.
```

Lines 14 through 25 take the GDSII file we just made and import it into a VlsiLayout object to play with in Python. The object is called “myLayout”. To do the actual import, we first create a Gds2Reader object called “reader” in line 19. Notice that “myLayout” is passed in to the reader for its constructor to use. In like 24, we tell “reader” to fill the layout object “myLayout” with data from our GDSII file by calling the method “loadFromFile” and passing it a valid path. At this point, the “myLayout” object knows everything about the shapes and properties of our layout and we are no longer in the world of GDS.

```
14. #create a layout object - this object represents all of the elements within the layout
15. myLayout = gdsMill.VlsiLayout()
16.
17. #give our layout object to a gds reader object.  The gds reader will look at the binary gds 2
    file and
18. #populate the layout object with the file's contents
19. reader = gdsMill.Gds2reader(myLayout)
20.
21. #tell the reader object to process the gds file that we streamed in above
22. #un-comment the next line to see some details about the gds contents
23. #reader.debugToTerminal=1
24. reader.loadFromFile("./gdsFiles/testLayoutA.gds")
25.
```

Lines 26 through 38 perform some modification on our layout. As a simple example, we are just going to add a box to the layout using the addBox method. For the layerNumber, we simply tell it to use the first layer specified in the original layout. In a real example, you would specify a meaningful number here to correspond to a particular layer (i.e. 7 = metal1). Also notice that in line 32, the internal map is updated. This is because we don’t plan to add any other shapes to the layout, and we want to visualize it as a PDF later. Therefore, we want our newly added box to be updated in the internal flat map.

```
26. #our layout object now contains all of the elements of the layout
27. #let add a box to the layout
28. myLayout.addBox(layerNumber = myLayout.layerNumbersInUse[0],    #pick some layer
29.                 offsetInMicrons = (-10,0),  #location
30.                  width = 10.0,    #units are microns
31.                  height = 5.0,
32.                  updateInternalMap = True, #This is important for visualization - see note 1
    below
33.                  center = True)  #origin is in the center or the bottom left corner
34. #Note 1:the layout object keeps track of its elements in a 2D map (no hiearachy)
35. #this makes visualization more efficient.  Therefore, to do a PDF output or some other
36. #flat output, you need to "update the internal map" of the layout object.  Only do this
37. # ONE TIME after all the objects you are manipulating are fixed
38.
```

Lines 39 through 49 stream our modified layout back into Cadence. This first thing we do is change the name (if we didn't, our original layout might be overwritten). We then create a Gds2writer object, in exactly the same way as the Gds2reader earlier. Line 44 exports the layout to a GDS2 file, which we use in line 46 to stream back into Cadence.

```
39. #let's take out new layout and stream it back into a cadence library
40. #first, create a new GDS
41. #we change the root structure name (this will be the cellview name upon stream in)
42. myLayout.rename("testLayoutB")
43. writer = gdsMill.Gds2writer(myLayout)
44. writer.writeToFile("./gdsFiles/testLayoutB.gds")
45.
46. streamer.streamToCadence(cadenceLibraryContainerPath = "~/design/600nmAmi",
47.                          libraryName = "gdsMillTest",
48.                          inputPath = "./gdsFiles/testLayoutB.gds")
49.
```

Lines 50 through 73 create a PDF view of our modified layout. We first create a PdfLayout object called "visualizer" and we pass in our layout to its constructor. In lines 61 through 66, we assign hexadecimal colors to the first 6 layers of our layout. By using the layerNumbersInUse property, we don't need to know the actual layer numbers. The scale is set to 1/500 in line 69 to reduce the size of the PDF file. Lines 71 through 73 finally draw the layout and export it to a file.

```
50. #let's create a PDF view of the layout object
51. #first, create the object to represent the visual output
52. visualizer = gdsMill.PdfLayout(myLayout)
53.
54. #since we have no knowledge of what the layer numbers mean for this particular technology
55. #we need to assign some colors to them
56.
57. #uncomment the following line if you want to actually see the layout numbers in use
58. #print myLayout.layerNumbersInUse
59.
60. #for each layer number used in the layout, we will asign it a layer color as a RGB Hex
61. visualizer.layerColors[myLayout.layerNumbersInUse[0]]="#219E1C"
62. visualizer.layerColors[myLayout.layerNumbersInUse[1]]="#271C9E"
63. visualizer.layerColors[myLayout.layerNumbersInUse[2]]="#CC54C8"
64. visualizer.layerColors[myLayout.layerNumbersInUse[3]]="#E9C514"
65. visualizer.layerColors[myLayout.layerNumbersInUse[4]]="#856F00"
```

```
visualizer.layerColors[myLayout.layerNumbersInUse[5]]="#BD1444"

#set the scale so that our PDF isn't enormous
visualizer.setScale(500)
#tell the pdf layout object to draw everything in our layout
visualizer.drawLayout()
#and finally, dump it out to a file
visualizer.writeToFile("./gdsFiles/gdsOut.pdf")
```

Array Example

The array demo script is located in the exampleUserDir directory. It gives a step by step example of using GDS Mill to instantiate a Cadence layout inside of a newly created layout. The instances are placed in an array along with some top level shapes and texts. These steps demonstrate the true power of scripting layout generation with GDS Mill - tasks such as memory compilation, power grid generation, etc. can be implemented quickly with high extensibility.

**BE SURE TO EDIT THE PATHS IN THE EXAMPLE SCRIPT TO MATCH THE ACTUAL PATHS USED IN YOUR SYSTEM

Lines 1 through 24 stream the layout that we want to instantiate into a VlsiLayout object called “arrayCellLayout”. These steps were all covered in the Quick Start example.

```
1.  #!/usr/bin/env python
2.  import gdsMill
3.  #we are going to make an array of instances of an existing layout
4.  #assume that we designed the "base cell" in cadence
5.  #step 1 is to stream it out of cadence into a GDS to work with
6.  #   creater a streamer object to interact with the cadence libraries
7.  streamer = gdsMill.GdsStreamer()
8.
9.  #   use the streamer to take a cadence layout, and convert it to GDS 2 for us to work with
10. #   the GDS will be named testLayoutA.gds
11. streamer.streamFromCadence(cadenceLibraryContainerPath = "~/design/600nmAmi",
12.                            libraryName = "gdsMillTest",
13.                            cellName = "testLayoutA",
14.                            outputPath = "./gdsFiles")
15.
16. #next, load our base cell layout from the GDS generated above
17. arrayCellLayout = gdsMill.VlsiLayout()
18. reader = gdsMill.Gds2reader(arrayCellLayout)
19. reader.loadFromFile("./gdsFiles/testLayoutA.gds")
20.
21. ##since we will be streaming into the same library that testLayout came from
22. #let's rename it here so that we don't overwrite accidentally later
23. arrayCellLayout.rename("arrayCell")
24.
```

Lines 25 through 53 create the layout array. First, an empty top level layout is created in line 28. Notice that a name is assigned in the constructor, which will be the name of the final cellView when streamed back into Cadence. A nested loop is setup in line 43 and 44. Lines 46 through 48 are included to demonstrate mirroring that might be performed in a typical memory compiler. Every other instance is mirrored about its X axis. Line 49 performs the actual instantiation of “arrayCellLayout” into “newLayout”. At the end of the nested loops, there will be 100 instances placed in a grid spaced by 10 microns in the x direction and 15 in the y.

```
25. #now create a new layout
26. #be sure to assign a name, since this will be the root object in our hierarchy to which
27. #all other objects are referenced
28. newLayout = gdsMill.VlsiLayout(name="arrayExample")
29.
30. #now place an instnace of our top level layout into the filled layout
31. #hierarchy looks like this:
32. #    array example
33. #        array cell layout
34. #            layout elements
35. #            layout elements
36. #            layout elements
37. #        cell instance
38. #        cell instance
39. #        cell instance
40. #        connection elements .....
41.
42. #now create the array of instances
43. for xIndex in range(0,10):
44.     for yIndex in range(0,10):
45.         if(yIndex%2 == 0):
46.             mirror = "MX"
47.         else:
48.             mirror = "R0"
49.         newLayout.addInstance(arrayCellLayout,
50.                               offsetInMicrons = (xIndex*10.0,yIndex*15.0),
51.                               mirror = mirror,
52.                               rotate = 0.0)
53.
```

A typical layout design will need to connect the previously placed instances together in some way. Lines 54 through 66 create a path and a text element in the top level layout. They could be rolled into a loop and parameterized in a variety of ways to effectively create top level metallization with pins. Noticve that the internal map is updated in line 65 since this is the last element we add to the layout.

```
54. #add a "wire" that in a real example might be a power rail, data bus, etc.
55. newLayout.addPath(layerNumber = newLayout.layerNumbersInUse[7],
56.                   coordinates = [(-20.0,0.0),(25.0,0),(25.0,10.0)],
57.                   width = 1.0,
58.                   updateInternalMap = False)
59. #add some text that in a real example might be an I/O pin
60. newLayout.addText(text = "Hello",
61.                   layerNumber = newLayout.layerNumbersInUse[5],
62.                   offsetInMicrons = (0,0),
63.                   magnification = 1,
64.                   rotate = None,
65.                   updateInternalMap=True)
66.
```

Lines 67 through 73 perform the same GDS export and stream in operations as seen in the Quick Start example above.

```
67. #and now dump the filled layout to a new GDS file
```

```
writer = gdsMill.Gds2writer(newLayout)
writer.writeToFile("./gdsFiles/arrayLayout.gds")
#and stream it into cadence
streamer.streamToCadence(cadenceLibraryContainerPath = "~/design/600nmAmi",
                         libraryName = "gdsMillTest",
                         inputPath = "./gdsFiles/arrayLayout.gds")
```

Layer Fill Example

The filler demo script is located in the exampleUserDir directory. It gives a step by step example of performing a layer fill on top of an existing layout. This step is typically required towards the end of a VLSI design cycle to satisfy density requirements for fabrication.

**BE SURE TO EDIT THE PATHS IN THE EXAMPLE SCRIPT TO MATCH THE ACTUAL PATHS USED IN YOUR SYSTEM

Lines 1 through 9 perform the same setup to read in a GDSII layout as in the Quick Start example. We assume in this case the the gds has already been streamed out, either via GDS Mill, or the Cadence CIW.

```
1.  #!/usr/bin/env python
2.  import gdsMill
3.
4.  #we will add the filler at a higher level of hiearchy
5.  #so first, load our top level layout from GDS
6.  myTopLevelLayout = gdsMill.VlsiLayout()
7.  reader = gdsMill.Gds2reader(myTopLevelLayout)
8.  reader.loadFromFile("./gdsFiles/testLayoutA.gds")
9.
```

Since we don't want to add fill directly to our existing layout, we will create another level of hierachy on top of it. Lines 10 through 28 create a new top level layout and drop in an instance of the existing layout into it.

```
10. #now create a new layout
11. #be sure to assign a name, since this will be the root object in our hierarchy to which
12. #all other objects are referenced
13. filledLayout = gdsMill.VlsiLayout(name="filledLayout")
14.
15. #now place an instnace of our top level layout into the filled layout
16. #hierarchy looks like this:
17. #   filled layout
18. #       top level layout
19. #           layout elements
20. #           layout elements
21. #           layout elements
22. #       fill elements
23. #       fill elements .....
24. filledLayout.addInstance(myTopLevelLayout,
25.                                 offsetInMicrons = (0,0),
26.                                 mirror = "",
27.                                 rotate = 0.0)
28.
```

Lines 29 through 45 perform the actual fill operation by calling the "fillAreaDensity" method on the top level layout. We perform this action twice to fill two different layers using varied spacing, box size, etc.

```
29. #now actaully add the fill - gds mill will create an array of boxes
30. # maintaining spacing from existing layout elements
31. #we'll do it once for two different layers
32. filledLayout.fillAreaDensity(layerToFill = myTopLevelLayout.layerNumbersInUse[5],
33.                              offsetInMicrons = (-10.0,-10.0), #this is where to start from
34.                              coverageWidth = 40.0,  #size of the fill area in microns
35.                              coverageHeight = 40.0,
36.                              minSpacing = 0.5,  #distance between fill blocks
37.                              blockSize = 2.0 #width and height of each filler block in microns
38.                              )
39. filledLayout.fillAreaDensity(layerToFill = myTopLevelLayout.layerNumbersInUse[7],
40.                              offsetInMicrons = (-11.0,-11.0), #this is where to start from
41.                              coverageWidth = 40.0,  #size of the fill area in microns
42.                              coverageHeight = 40.0,
43.                              minSpacing = 0.5,  #distance between fill blocks
44.                              blockSize = 3.0 #width and height of each filler block in microns
45.                              )
```

Lines 46 through 53 dump the top level filled layout to a GDS file and stream it back into the original Cadence library. Refer to the Quick Start example for more details.

```
46. #and now dump the filled layout to a new GDS file
47. writer = gdsMill.Gds2writer(filledLayout)
48. writer.writeToFile("./gdsFiles/filledLayout.gds")
49. #and strea
50. streamer = gdsMill.GdsStreamer()
51. streamer.streamToCadence(cadenceLibraryContainerPath = "~/design/600nmAmi",
52.                           libraryName = "gdsMillTest",
53.                           inputPath = "./gdsFiles/filledLayout.gds")
```

Class Reference

GdsPrimitives Module

This module contains a group of class definitions for the various elements that comprise a GDSII file. These include *structures, boundaries, paths, structure references, array references, text, nodes, and boxes*.

Relevant methods:

rotatedCoordinates(rotateAngle):

- Implemented in *GdsBoundary*, *GdsPath*
- For geometric elements whose coordinates are specified relative to their origin, this helper method rotates these coordinates by an arbitrary angle.
- Arguments: *rotateAngle* is a numeric (float / int) angle in degrees
- Returns: A list of (x,y) pairs representing the rotated coordinates

equivalentBoundaryCoordinates(rotateAngle):

- Implemented in *GdsPath*
- Since a path's coordinates only define its points along it's center, this method uses the *pathWidth* property to convert the path coordinates into a list of boundary coordinates. This list effectively draws the outline of the path and is used for visualizing the path, for example as a PDF element.
- Arguments: *rotateAngle* is a numeric (float / int) angle in degrees
- Returns: A list of (x,y) pairs representing the rotated boundary-equivalent coordinates

Gds2reader Class

Relevant methods:

__init__(layoutObject, [debugToTerminal]):

- Description:
  - Constructor method.
- Arguments:
  - *layoutObject* is an object of type *VlsiLayout* that the reader will populate with information from a GDSII file.
  - *debugToTerminal* is an optional parameter that can be set to 1 or 0, where 1 will print GDS specific information on the terminal as files are read
- Return: None

loadFromFile(fileName):

- Description:
  - When called, reads the contents of a binary GDSII file and populates the attached *VLSILayout* object with *GDSPrimitives* objects.
- Arguments:
  - *fileName* is an absolute string path to a binary GDS 2 file.
- Return: None

Gds2writer Class

Relevant methods:

__init__(layoutObject):

- Description:
  - Constructor method.
- Arguments:
  - *layoutObject* is an object of type *VlsiLayout* that the write will use to generate a binary GDSII file.
- Return: None

writeToFile(fileName):

- Description:
  - When called, writes the elements of a *VLSILayout* object to a binary GDSII file.
- Arguments:
  - *fileName* is an absolute string path to the binary GDS 2 file to be created.
- Return: None

VlsiLayout Class

Relevant properties:

units:

- Description:
  - A tuple that maps layout units to physical units.
- Default:
  - (0.001, 1e-9) which corresponds to 1 layout unit = 1 micron

layerNumbersInUse:

- Description:
  - A list of all layer number currently in the layout.

rootStructureName:

- Description:
  - A string holding the name of the top most structure in the layout hierarchy. This is also used as the cellView name upon import / export to Cadence. Change this only with the *rename* method.
- Default:
  - None

Relevant methods:

__init__(self, [name], [units], [libraryName], [gdsVersion]):

- Description:
  - Constructor method.
- Arguments:
  - *name* is a string that can be used to initialize the rootStructure name. (See rootStructureName above)
  - *units* is a tuple that maps layout units to physical units.

    - *libraryName* is the name of the Library that the layout was generated from. Irrelevant for streaming back into Cadence.
    - *gdsVersion* is the file format version for GDSII. Default is 5.
- Return: None

rename(newName):

- Description:
    - Changed the name of the root structure object. This will effectively rename the "top level" of the layout which effects the cellView name upon import to Cadence.
- Arguments:
    - *newName* is a string holding the new top level name.
- Return: None

addInstance(layoutToAdd, [offsetInMicrons], [mirror], [rotate], [updateInternalMap]):

- Description:
    - Adds an instance of *VlsiLayout* to the current layout. This allows one to create hierarchy within generated layouts.
- Arguments:
    - *layoutToAdd* is a *VlsiLayout* object describing the layout to instantiate.
    - *offsetInMicrons* is a *tuple* of (x,y) coordinates in layout units that sets the position of the instantiated layout. Default is (0,0).
    - *mirror* is a *string* of GDS mirror properties. Possible values are "x", "y", "xy". Default is None.
    - *rotate* is a *float* containing the angle of rotation. Common values are 180.0 and 90.0. Default is None.
    - *updateInternalMap* is a *boolean* specifying whether or not to update the internal flat representation of the layout. This flat version is required for visualization, such as exporting to PDF, and for certain geometric operations such as density filling. For a given layout, this internalMap only needs to be updated once. Default is False.
- Return: None

addBox(layerNumber, offsetInMicrons, width, height, updateInternalMap, center):

- Description:
    - Adds a geometric rectangle to the layout using the given parameters.
- Arguments:
    - *layerNumber* is an integer value corresponding to a particular layout layer. Refer to *layerNumbersInUse* to see which layer numbers are currently used in the layout. Any valid layer is acceptable.
    - *offsetInMicrons* is a *tuple* of (x,y) coordinates in layout units that sets the position of the generated rectangle. Default is (0,0)
    - *width* is the *float* width of the rectangle in layout units.
    - *height* is the *float* height of the rectangle in layout units.
    - *updateInternalMap* is a *boolean* specifying whether or not to update the internal flat representation of the layout. This flat version is required for visualization, such as exporting to PDF, and for certain geometric operations such as density filling. For a given layout, this internalMap only needs to be updated once. Default is False.
    - *center* is a *boolean* determining if the offset is relative to the center of the rectangle or the bottom left corner. Default is False.
- Return: None

addPath(layerNumber, coordinates, width, updateInternalMap):

- Description:

    - Adds a geometric path to the layout using the given parameters.
- Arguments:
    - *layerNumber* is an integer value corresponding to a particular layout layer. Refer to *layerNumbersInUse* to see which layer numbers are currently used in the layout. Any valid layer is acceptable.
    - *coordinates* is a list of *tuples* of (x,y) coordinates in layout units that sets the shape of the generated path.
    - *width* is the *float* width of the path in layout units.
    - *updateInternalMap* is a *boolean* specifying whether or not to update the internal flat representation of the layout. This flat version is required for visualization, such as exporting to PDF, and for certain geometric operations such as density filling. For a given layout, this internalMap only needs to be updated once. Default is False.

addText(text, layerNumber, offsetInMicrons, magnification, rotate, updateInternalMap)
- Description:
    - Adds a geometric text string to the layout using the given parameters.
- Arguments:
    - *layerNumber* is an integer value corresponding to a particular layout layer. Refer to *layerNumbersInUse* to see which layer numbers are currently used in the layout. Any valid layer is acceptable.
    - *offsetInMicrons* is a *tuple* of (x,y) coordinates in layout units that sets the position of the generated text. Default is (0,0)
    - *magnification* is the *float* size of the generated text. Default is 0.1.
    - *rotate* is a *float* containing the angle of rotation. Common values are 180.0 and 90.0. Default is None.
    - *updateInternalMap* is a *boolean* specifying whether or not to update the internal flat representation of the layout. This flat version is required for visualization, such as exporting to PDF, and for certain geometric operations such as density filling. For a given layout, this internalMap only needs to be updated once. Default is False.
- Return: None

fillAreaDensity(layerToFill , offsetInMicrons, coverageWidth,
coverageHeight, minSpacing, blockSize):
- Description:
    - Adds many blocks of a particular layer to the layout while maintaining a specified distance from any existing shapes. Will take into account all levels of hierarchy even when invoked at the highest level of hierarchy.
- Arguments:
    - *layerToFill* is an integer value corresponding to a particular layout layer. Refer to *layerNumbersInUse* to see which layer numbers are currently used in the layout. Any valid layer is acceptable.
    - *offsetInMicrons* is a *tuple* of (x,y) coordinates in layout units that allows nudging of the filled layout by specifying the center of the area to be filled.
    - *coverageWidth* and *coverageHeight* are the *float* dimensions of the area to be filled centered around the offset above.
    - *minSpacing* is a *float* of layout units to specify the minimum distance from a generated block to an existing shape.
    - *blockSize* is a *float* of layout units to specify the size (square) of the generated blocks.
- Return: None

PdfLayout Class

This class is used for visualization of VlsiLayout objects. It makes use of the PyX framework to translate GDS Primitives into lines, shapes, and colors in PDF format.

Relevant properties:

layout:

- Description:
  - A *VlsiLayout* object to be drawn in the PDF.

layerColors:

- Description:
  - A dictionary whose keys are layer numbers and values are hex strings representing RGB color values.
  - Be sure to populate this dictionary for at least 1 layer.
  - For example: layerColors[7] = “#FFFF”
- Default:
  - Empty

scale:

- Description:
  - A float that scales the size of the output PDF.
  - Since most layouts are thousands of layout units in size, coordinates are DIVIDED by this scale value in order to keep the PDF size reasonable.
- Default:
  - 1.0

Relevant methods:

__init__(theLayout):

- Description:
  - Constructor method.
- Arguments:
  - *theLayout* is the VlsiLayout object to be converted to PDF format.
- Return: None

drawLayout():

- Description:
  - Once everything else is setup (colors assigned, layout connected) this method goes through and draws the actual layout shapes into an internal buffer.
- Arguments: None
- Return: None

writeToFile(fileName):

- Description:
  - Outputs a PDF version of the layout to the specified filename.
- Arguments:
  - *fileName* is the string path to the PDF file to be generated.
- Return: None