ece 260c. **Lab 2** Scripting with the Database

**Your name / Your PID**

Please enter your name and PID above – you agree to the academic integrity and course policies outlined in the [Syllabus](https://abkcourses.github.io/ece260c/docs/broken). Please ensure you submit this lab within this report document template.

This lab will continue to use the same Docker container as the previous labs. **Please ensure GUI is enabled** - see the [Software Setup Guide](https://abkcourses.github.io/ece260c/docs/tools) if you are unsure.

GitHub Classroom for accessing all necessary data files and submitting your work.

[Click here to unlock the assignment in GitHub Classroom.](https://abkcourses.github.io/ece260c/docs/broken)

Then run,

| gh repo clone ABKCourses/ece260c-lab2-YourUsername  cd ece260c-lab2-YourUsername |
| --- |

# Section I Working with Python in OpenROAD

The database is the core of OpenROAD – and of any EDA tool. It stores both design and technology data in a unified object model that lives in memory and on disk. By giving every OpenROAD component access to this common, [callback-enabled model](https://vlsicad.ucsd.edu/Publications/Conferences/374/c374.pdf) (as seen in classroom lectures), the tool stays modular while still supporting tight, iterative optimization loops.

Traditionally, EDA tools treat the user as an after-thought in the data model. Accessing or modifying design data typically requires one of three awkward routes:

* **GUI interaction:** slow, manual, and error-prone.
* **Tcl scripting:** a niche language with limited libraries and an idiosyncratic syntax.
* **File export/import:** writing to an open format, then re-reading it – an expensive round-trip that depends on intricate standards and often loses information (e.g., detailed timing annotations).

Each of the above options hampers productivity and makes advanced customization difficult. OpenROAD is unique in that it introduces two great improvements to this status quo. First, being open source means the full OpenDB source is available to engineers needing custom components or the full breadth of data – with the class definitions serving a self-documenting purpose. Major components like the global placer being directly modifiable also help enable bespoke changes.

The second breakthrough is native Python support inside OpenROAD. Engineers can now tap the entire Python ecosystem—NumPy, pandas, PyTorch, and more—to query and manipulate OpenDB objects directly, side by side with the traditional Tcl commands. This makes physical-design scripting far more expressive and also enables ML-driven workflows: tools such as NVIDIA’s [CircuitOps](https://github.com/NVlabs/CircuitOps) use the Python API to [stream circuit structure and metrics from OpenDB](https://vlsicad.ucsd.edu/Publications/Conferences/407/c407.pdf) straight into machine-learning frameworks, eliminating slow file round-trips and extra conversion code.

In this section, you will begin to explore the Python interface by using it to manipulate the database.

One thing to note is that OpenROAD currently does not support running the GUI alongside Python so we will first run our scripts to generate our results and then, if needed, we will reopen these results in the GUI.

## Anatomy of a Script

Open template.py in your repository; it shows the canonical structure of an OpenROAD Python script. Every OpenROAD Python script will typically follow this basic format. The first lines import OpenROAD”s Tech/Design objects as well as OpenDB’s own bindings to Python (odb), which can be used to create database objects but is more experimental. Tech and Design expose the same global OpenDB instance that OpenROAD uses when you launch the tool normally. In other words, *OpenROAD – not your script – owns the database*. Your Python code merely holds references to objects that already live inside that global DB.

Consequences of this ownership model:

* **Objects persist after a function returns.** If you load a design inside a helper function and then exit that function, the design remains in the database. Attempting to load another copy of the same block later will trigger name collisions unless you delete the original instance first.
* **Dangling references are possible.** Deleting an object from the DB does *not* invalidate any Python variables that still point to it. Accessing or modifying such a stale handle can crash OpenROAD (e.g., seg-fault).

Rule: treat the Python API with the same discipline you would apply in Tcl or C++: load only what you need, delete what you discard, and avoid keeping stray references around.

### The Design Object

The Design object we create is our interface to OpenROAD. It contains many methods but the important ones can be classified into the following categories:

| **Function** | **Uses** |
| --- | --- |
| getReplace, getTritonroute, … | These getters correspond to their respective components of the OpenROAD application. Once you have these objects, you can control this component.  For example, design.getGlobalRouter().globalRoute() will run global routing. |
| getDb | Returns the *dbDatabase* object – OpenROAD’s master OpenDB instance.  While the getBlock method described below is more convenient for getting your actual design database, use this when you need low-level access (e.g., LEF/DEF or Liberty tech/library data). |
| getBlock | Returns the primary *dbBlock* within the database (i.e., the top-level design), allowing you to access its instances, etc.  Equivalent to design.getDb().getChip().getBlock() but shorter. Note that a DB can hold multiple chips to support multi-die designs. |
| evalTclString | Executes an arbitrary Tcl command string and returns its result.  This can be useful in situations where Python bindings lack a one-to-one wrapper. E.g., design.getReplace() does not have the same functionality as the familiar global\_placement Tcl command. |
| createDetachedDb | Creates a stand-alone *dbDatabase* that is **not** connected to the global OpenROAD DB. This allows you to load and manipulate multiple designs side-by-side. *(Experimental – use with care.)* |
| readDb, readDef, readVerilog  writeDb, writeDef | Commands for reading and writing database/design snapshots and netlists |

## Digging in Interactively

Let’s quickly explore the database format. In the terminal, cd into section1/ and run   
openroad -python template.py. You will be dropped into the interactive Python REPL. Ignore the warning “.openroad ignored with -python”.

If you look at template.py, you will see that we have already run readDb and getBlock, giving us the convenient database dbBlock object in a variable called block.

**Q1.1** Let’s use Tab completion to explore the block. Type in block. (with the dot) and press Tab – you should see the member methods of block. Paste a screenshot below.

What function might you use to get the area (bounding box) of the design? What about the pins? (refer to the Databases lecture)

If you now try to **Tab-complete** on a function like block.getInsts(), you will get no output. This is because Python cannot predict the output of function calls. Instead, we need to store the intermediate output of this function so we can explore its methods.

**Q1.2** Run insts = block.getInsts(). This will return an array of instances. Then, try Tab completion on insts[0]. You will see that this doesn’t work because of the array indexing. You could make a new variable to Tab-complete on but, even better, you can run dir(insts[0]). Run this command, then paste a screenshot below.

Run exit() to leave the OpenROAD interactive Python REPL.

## Writing your First Script – Cell Area

Now that you have some idea of the database’s structure, you can try to write your first Python script.

Use the cp command to create a new file based on template.py called area.py. This will be our new script for retrieving the sum total area utilized by cells.

In this script, use getInsts as you did above to get all instances in the block. To get areas, however, we will need their masters. If you remember from our lecture on Databases, masters are the DB term for the actual standard cell designs from the PDK – intended to be replicated across the design.

**Q1.3** After inspecting the dbInst API in Q 1.2, identify the method you would call on a single instance to obtain its master cell (dbMaster).

Retrieving masters for an entire array of instances in a purely procedural language like Tcl or C requires effort to loop through each instance and retrieve its master one by one. Python instead provides a convenient functional syntax known as [List Comprehensions](https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions).   
*Hint: once you know the method, you can collect all instances with a list comprehension like in this example:* [i.getName() for i in inst] # Returns an array of cell names

**Q1.4** Write a list comprehension for retrieving all masters into a variable called masters. Place this line both into your script and also below.

Using the interactive REPL investigation techniques you learned above, run your current area.py in OpenROAD and try to find the function for getting the area of a master.

**Q1.5** Using the function you found, write a list comprehension for retrieving cell areas per master into a variable called areas. Place this line both into your script and also below.

Now that we have the areas of each cell, we need to sum them up. Rather than looping, Python provides a built-in function called sum that will add together all elements in an array.

**Q1.6** Use sum to find the total area, then print it to the console. Run the script with openroad -exit -python area.py. Paste a screenshot of the output below.

What you may notice is that the area number you got is an incredibly large integer. This number is not because it’s not necessarily represented in any standard unit, but rather in what is known as Database Units (DBUs). Integers have historically been used because of both precision and performance considerations – you can read about datatype considerations [here](https://openroad.readthedocs.io/en/latest/contrib/DatabaseMath.html).

DBUs have no relation to live manufacturing units except as defined in the PDK’s [technology LEF](https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/blob/11b93cfa1d9c61edd4a9e45322c8fa2839c24f7b/flow/platforms/ihp-sg13g2/lef/sg13g2_tech.lef#L25), which contains a DATABASE MICRONS directive that assigns a certain ratio of DBUs to micron (1000 is common).

Thus, to find how many microns a certain distance is, its length in DBUs must be divided by this ratio. Keep in mind that, for area, the unit is DBU2, **so the ratio also needs to be squared**.

**Q1.7** Using the ratio already provided in the variable dbu\_per\_micron, now print out the area in um2 – it’s best practice to also print out the unit. Run the finished script and paste a screenshot of the output below.

Your area.py is now complete. Make sure you commit this completed version to your GitHub repository.

# Section II Mutating the Database

Until now you’ve only *read* information from the database. In the following exercises, you will learn how to *modify* it – specifically by changing cell placements. (We will limit the scope to placement objects and skip topics like dbWire or dbNet for now.)

## Swapping Masters

In section2/, Use the cp command to create a new file based on section2/template.py called swaps.py. Here, we will write a script to swap masters for all instances of a given master.

In practice, swapping masters is something that the flow does during the resizing pass – the pass where, after global placement, buffers are inserted and cells are sized up (down) to meet drive requirements. For this exercise, we will indiscriminately size up all instances of a certain master – sg13g2\_dfrbp\_1 – a positive-edged DFF.

Using what you have learned in Section I, retrieve all instances. With these loaded, we can take advantage of another feature of list comprehension – filtering. You can use the following line to filter out only instances that are of master sg13g2\_dfrbp\_1:

| filtered\_insts = [i for i in insts if i.getMaster().getName() == "sg13g2\_dfrbp\_1"] |
| --- |

This syntax makes it possible to perform filters and, if needed, manipulations at the same time.

Now, we need to actually perform the swap. We will be swapping from sg13g2\_dfrbp\_1 to sg13g2\_dfrbp\_2, a higher strength, larger cell. Before we can do this, we need to retrieve this new master. We could do this by getting all masters in the library and then filtering out the one with the correct name but the dbLibrary object provides us with the convenient findMaster method. Try the following:

| library.findMaster("sg13g2\_dfrbp\_2") |
| --- |

With the master loaded, it’s now possible to use the swapMaster function on the dbInst’s.

You could use a loop with the for i in insts: syntax or you can use an unfiltered list comprehension as before.

With the script completed, save it and run it. You should see a new file generated called swaps.odb.

In your terminal, run openroad -gui and, when the Scripting pane at the bottom comes up, run:

| read\_db swaps.odb |
| --- |

**Q2.1** Take a screenshot of the OpenROAD GUI window and paste it here. Take note of the overlapping cells.

This overlap problem is something that the normal flow resizer has as well – the thing to take note here is that changes you make to the DB are not necessarily physically or logically valid. The DB itself has no notion of placement/routing legality and thus, it has no enforcement. Depending on what kind of DB manipulation you are doing, you will have to deploy various remedies. For small changes to placed cells like in this case, we probably want to run the legalizer (detailed placer – DPL in OpenROAD).

Close the OpenROAD GUI and add the following to your script:

| design.evalTclString("detailed\_placement") |
| --- |

**Q2.2** Run the latest script, then reopen the OpenROAD GUI, load the updated swaps.db, and paste the screenshot below.

Your swaps.py is now complete. Ensure the latest swaps.odb is present as well. Make sure you commit these completed files to your GitHub repository.

## Moving & Locking Placements

Use the cp command to create a new file based on template.py called move.py. Here, we will write a script to move a single cell around. We will use the dbInstance setLocation(x, y) method. The origin (0, 0) is at the bottom left of the chip so increasing the coordinates in the positive dimension moves objects toward the top right – remember also that this is all expressed in DBU.

Using what you know above, write a script to select all sg13g2\_dfrbp\_1 instances. Now, use setLocation to place them at (50000, 50000) – roughly the center of this chip.

Now, legalize it using the evalTclString invocation you learned above.

**Q2.3** Run this script and open the new move.odb in the OpenROAD GUI. Paste a screenshot below. Can you see the instances we moved?

Now, we have this specialized placement of our DFFs but the rest of our design is no longer really congruent with it – we want to rerun placement on the rest of the design. The detailed placement relies on a good estimate from global placement so it is important that we rerun GPL again. After our first detailed placement, which generated a legal placement of our DFFs, write the following command to execute a full placement:

| design.evalTclString("global\_placement; detailed\_placement") |
| --- |

**Q2.4** Run this script and open the updated move.odb in the OpenROAD GUI. Paste a screenshot below. What do you think happened to our FFs in the placement process? Answer in 1-2 sentences.

The database does not track “user overrides.” When you manually change a cell’s location and then invoke RePlAce (or any other placer), the tool freely overwrites that location because it has no flag telling it to keep the cell fixed. To solve this problem, we can use Placement Status.

**Q2.5** Every instance has a placement status and it is possible to find it in the GUI within the Inspector tab. Keep the GUI open on move.odb, select any cell, and paste a screenshot of the Inspector window below. What is its Placement Status set to?

Then, triple-click on this value to bring up a drop-down of possible statuses – what can you see?

For our purposes, we care about two placement statuses, PLACED and LOCKED.

This time, before the line executing global/detailed placement commands (but after the first call to the detailed placer) in your Python script, add a call to the setPlacementStatus("LOCKED") method on all sg13g2\_dfrbp\_1 instances.

**Q2.6** Run this script and open the updated move.odb in the OpenROAD GUI. Select a DFF cell and then paste a screenshot below. Is its placement correct now? Can you see its updated placement status? Answer in 1-2 sentences.

Your move.py is now complete. Ensure the latest move.odb is present as well. Make sure you commit these completed files to your GitHub repository.

# **Section III** Writing Custom Placers

Now that you’ve explored Python scripting and the OpenDB data model, it’s time to put those pieces together. In the next section you’ll build a **custom placer**—a Python script that reads the current design, applies your own placement logic, updates the database, and then saves the new layout. This exercise will reinforce everything you’ve learned so far while showing how to combine querying, editing, and placement-status control in a single, practical tool.

## Why we Script our own Placers

By now, you’re familiar with the flat placement flow in OpenROAD, where **RePlAce (gpl)** handles the placement of both standard cells and macros. You've also seen **Hier-RTLMP (mpl)**, OpenROAD’s hierarchical placer that offers smarter macro placement by leveraging design hierarchy.

However, industry experience shows that automated placers like these often fall short when targeting the **highest-performance** or **highest-density** designs. Beyond managing deep hierarchies of functional blocks, real-world designs also employ **Structured Datapaths (SDPs)**—groups of standard cells or small macros placed tightly together to optimize dataflow physically.

To handle such requirements, we often rely on **custom SDP scripts**. These scripts allow fine-grained control over parameters such as **aspect ratio** and **design regularity**, enabling us to optimize placement quality for datapath-heavy blocks—similar to what we explored in **Lab 1**.

Another category of specialized placer is the **memory compiler**, which handles the placement of prebuilt memory tiles like **SRAM**, **ReRAM**, **eFuse**, and **flash**. These tools are typically proprietary and generate **LEF** files directly, rather than operating within a shared EDA database like OpenDB.

## Your Task

You will build a simple placer for a 128x1-bit Content-Addressable Memory (**CAM**) IP. This memory uses D Flip-Flops connected to XNORs so the memory does not output the data bit directly but rather a match bit. This is a simplistic example but is sufficient to see how SDPs might be placed.

Your goal is to place all DFFs in **a single column** and then all XNORs in a single column **to the right of the DFFs**. Then, use what you learned in Section II to perform legalization and then a separate full placement (including global placement) on the rest of the cells. Finally, you will perform basic routing.

The major requirement is that the placement be done such that **all elements must remain within the core area**, so that detailed placement later does not move cells too far, undoing your work.

**Q3.1** Before you begin, take a look at section3/design/section3.v. What do you notice about the style of Verilog? How is logic, particularly the logic we want to perform custom placement on, declared? How many total cells do we have as part of our custom placement? Answer in 2-3 sentences.

**Q3.2** Open section3.odb in the OpenROAD GUI and click around until you find cells that are part of the “cam” prefix. What exactly is their naming scheme like? (**Hint:** Try to focus on the naming patterns)

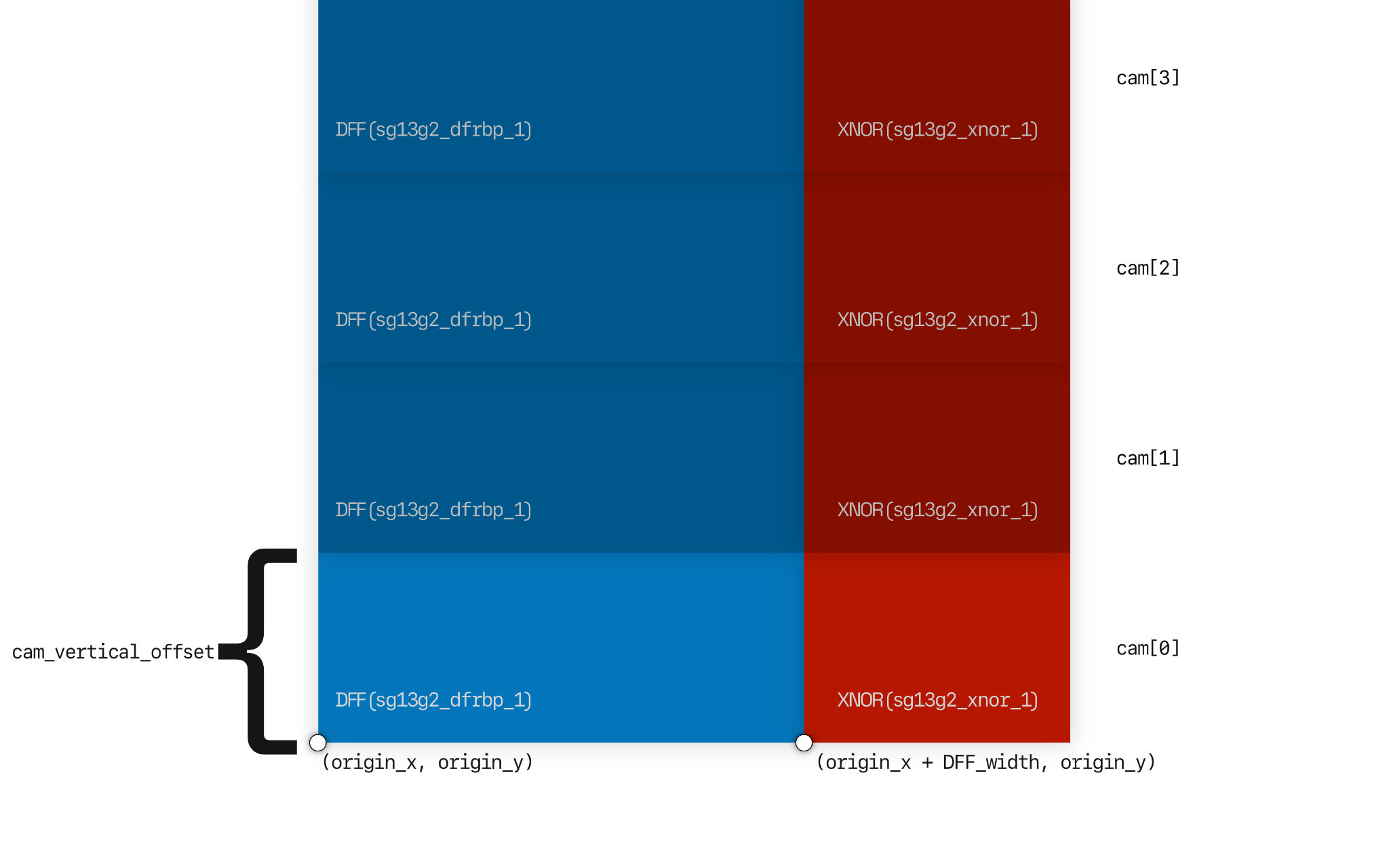
**Q3.3** From what you saw in Q3.2, construct a filter that you can use to find only the cells that are part of the CAM structure. Write it below as a list comprehension.

**Task** Using what you learned from the previous solutions, complete the script section3/placer.py.

Take note of the following:

* Use the predefined variables origin\_x and origin\_y as the base location of the DFF of cam[0] – refer to the diagram below
* You are provided the decode\_cam\_index(name) function as a utility for scripting your placer.
* Use the provided cam\_vertical\_offset variable to apply the correct vertical offset based on the cam index.
* Remember that your placement needs to be a tight packing – the XNOR should be exactly up against the right side of the DFF – refer to the diagram
* After the first round of legalization, remember to lock only these cam instances before the next round of global/detailed **placement** and then global/detailed **routing**.

**Refer to the diagram below:**



**Q3.4** When you have completed the placer, run the script, open the GUI, and take a screenshot of placer.odb and paste it here.

Your placer.py is now complete. Ensure the latest placer.odb is present as well. Make sure you commit these completed files to your GitHub repository.

**<End>**