In Aphyds we concentrate on the physical design of standard cell circuits. However, most chips will actually have multiple blocks, such as RAMs, datapaths, Intellectual Property (IP) blocks, and other elements. In order to lay out the entire chip, we must figure out where the standard cells go in relationship to these other items, that is floorplan the chip.

We will now develop a floorplanning algorithm for your chips, extending the Aphyds structure. The floorplanner is given a hierarchical (slicing tree) floorplan for the circuit, where the nodes may have multiple sizes. Your algorithm will perform the variable node size algorithm to determine the set of useful floorplans at each level in the slicing tree.

**Algorithm Description**

We assume that we have a slicing tree, which is a hierarchical floorplan where all non-leaf nodes have exactly two children. The idea is that each internal node represents the joining of the two sub-floorplans either vertically or horizontally at a slicing line.

We start with children representing the basic units to be floorplanned. Some are fixed, and thus have only one size \((W, H)\), where \(W\) is the width and \(H\) the height. Some blocks may be rotateable, which means they have two sizes: \((X, Y)\) and \((Y, X)\). Some may be more flexible, with multiple possible sizes.

We organize these sizes carefully. First, we make sure there are no redundant sizes. A redundant size would be a size \((X, Y)\) for an element, where another size \((W,H)\) exists such that \(W\leq X\) and \(H\leq Y\). Thus, this second layout is at least as small in width and height as \((X, Y)\), and there would be no reason to use \((X, Y)\) instead of \((W, H)\).

For each node in the slicing tree, we organize the potential sizes in increasing order of width, and thus decreasing order of height, since all redundant sizes are eliminated. Now, starting from the leaves up, we can calculate all of the legal, non-redundant sizes efficiently. Our algorithm will have the following subroutines:

- **makeVerticalSplits**: Compute all of the non-redundant floorplans that merge the sub-floorplans with a vertical split line (i.e. place one child next to the other horizontally). We do this by first forming a floorplan with the floorplan from each child with the smallest width. This gives the thinnest, tallest non-redundant vertical split floorplan. Now, any other non-redundant floorplan for this node must be shorter, since we produced the thinnest floorplan already. So, we throw away whichever child’s floorplan is the tallest (or both, if they are the same height), and form a floorplan with the remaining children’s sub-floorplans that have the minimum width. This process repeats until no floorplans remain for one of the children, at which point we are done.

- **makeHorizontalSplits**: Compute all of the non-redundant floorplans that merge the sub-floorplans with a horizontal split line (i.e. place one child above the other vertically). This is identical to makeHorizontalSplits, but it starts by creating the shortest, widest non-redundant floorplan for this node, then throws away the child floorplan which is the widest. To make things simple the input lists are given in increasing height order, and produced in increasing height order.

- **mergeFloorplans**: Once makeVerticalSplits and makeHorizontalSplits is done you have two sets of floorplans for this node. However, floorplans in one list may make floorplans in the other redundant. We solve this by merging the two lists. We start with lists in increasing width order. To merge, if one floorplan makes the other redundant, throw the redundant one out. If not, add the thinner one to the set of merged floorplans, and continue merging the remainder of the two lists.

Your job in this programming assignment will be to write these three routines. Note that makeVerticalSplits and makeHorizontalSplits will be virtually identical – write one, and the other will be fairly trivial to create.
Getting Started

You will build upon the code you produced in homework #2 – when the class is done, you’ll have a completely integrated toolsuite. Again, to run the program you compile and run “CircuitViewer.java”. Then, open a circuit (File->open) and start floorplanning (Edit->Floorplan Circuit). The slicing tree for your floorplanner is displayed in the upper left. Nodes are color-coded, with their size(s) displayed. Clicking on them will show their size combinations at the bottom. Note that the bottom pictures are scaled up individually to make them more readable – although some floorplans might thus seem redundant, when you check the sizes (displayed below their picture) you will see than none will be.

Once your floorplanning algorithm executes, you will fill in the internal nodes of the floorplanning tree. Right now they are shown by largely blank lines in the slicing tree, with a default icon. Note that these nodes also have handles for collapsing their children, helpful in navigating the display. Also, all nodes (leaves and internal) are given a unique letter as their designation – this will help in identifying specific nodes.

The slicing tree represents each node via a shorthand notation. Leaf nodes are in the form:

A: RAM <N (W1xH1)> …

A is the letter identifying this specific leaf node, followed by a type (what the node represents). Then, you will have one or more sizes, of the form <N (WxH)> where “N” designates a leaf node, and the width and height are given for this possible layout of the element.

Internal nodes are in the form:

J: <V A(WaXHa) B(WbxHb) = (Wv,Hv)> …

Here, we start with the letter designating this node. Then, there are one or more floorplans given. The first letter of the floorplan is a V (for Vertical split line) or H (for Horizontal split line). Then, the name and sizes used for the two children are given, where these children can be leaf or internal nodes. Specifically, in this example we merge together node A’s floorplan of size (Wa,Ha) with node B’s floorplan of size (Wb,Hb), to achieve a resulting overall floorplan (Wv,Hv). If this is confusing to you, pay attention to the bottom region of the dialog when you click a given slicing tree member – it graphically shows all of this information.

When your algorithm is complete, the final floorplan will be displayed in the upper right, along with the resulting sizes. Each node can be displayed individually in the bottom region, both leaves and internal, by clicking on them in the slicing tree. These tools should help you test and examine your algorithm’s behavior.

Programming Assignment

In this programming assignment you will need to implement the makeVerticalSplits, makeHorizontalSplits, and mergeFloorplans routines in SlicingTree.java. Again, you should look at the SlicingTree.html page produced by Javadoc to understand what needs to be done, and what routines will be useful in your efforts.

The Floorplan data structure will be important in your algorithm. It represents a single, possible layout for the elements within a subtree of the hierarchy. For the leaves (the basic elements in the system) this represents one of possibly multiple layouts for this element. For the internal nodes, this represents a floorplan for all of the elements within the subtree rooted at this node. You will primarily use the getSize() routine of Floorplan.java, as well as the constructor for floorplans for internal nodes (i.e. not the leaf floorplan constructor, since the floorplans for all leaf cells is specified in advance).

The routines checkFloorplans and FloorplansToString will be helpful in debugging – they will not need to be used in your production code, but will help get that code running. CheckFloorplans takes a Vector of floorplans and returns True if it is well-formed. If your Vectors don’t pass checkFloorplans you probably made a mistake (note: read the Javadoc help for makeHorizontalSplits for a special constraint on checkFloorplans within this routine).
FloorplansToString is a helper routine, which can help print out the current status of a Vector of Floorplans. If you need the state of such a Vector, do:
System.out.println(FloorplansToString(TheVectorToTest)); System.out.flush();

To run your program, open the floorplanner dialog box (described above), and then push the button. It will create floorplans for the given slicing tree. Once it is running (but perhaps not completely working), you should work bottom-up to verify your algorithm. Select an internal node connected directly to two leaf nodes, and check by hand if the floorplans are correct. If they are, collapse this node in the slicing tree, and continue testing. You can trace up the slicing tree to check each floorplan based upon already checked sub-floorplans to verify that things are working correctly, or to find the earliest failure point.

This programming assignment will require understanding the public and protected functions (those that appear in the .html files from Javadoc) for SlicingTree.java and Floorplan.java. You should only need to actually modify makeVerticalSplits(), makeHorizontalSplits(), and mergeFloorplans().

Experiments

Once your algorithm is running, you should take a look at the resulting floorplans. Notice the variation in final floorplans created for the circuit – there are many different potential solutions, often involving VERY different floorplans for the lower-level elements.