Magic Tutorial #4: Cell Hierarchies

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection

Commands introduced in this tutorial:

:array, :edit, :expand, :flush, :getcell, :identify, :load, :path, :see, :unexpand

Macros introduced in this tutorial:

x, X, ^X

1 Introduction

In Magic, a layout is a hierarchical collection of cells. Each cell contains three things: paint, labels, and subcells. Tutorial #2 showed you how to create and edit paint and labels. This tutorial describes Magic's facilities for building up cell hierarchies. Strictly speaking, hierarchical structure isn't necessary: any design that can be represented hierarchically can also be represented "flat" (with all the paint and labels in a single cell). However, many things are greatly improved if you use a hierarchical structure, including the efficiency of the design tools, the speed with which you can enter the design, and the ease with which you can modify it later.

2 Selecting and Viewing Hierarchical Designs

“Hierarchical structure” means that each cell can contain other cells as components. To look at an example of a hierarchical layout, enter Magic with the shell command **magic tut4a**. The cell **tut4a** contains four subcells plus some blue paint. Two of the subcells are instances of cell **tut4x** and two are instances of **tut4y**. Initially, each subcell is displayed in *unexpanded* form. This means that no details of the subcell are displayed; all you see is the cell’s bounding box, plus two names inside the bounding box. The top name is the name of the subcell (the name you would type when invoking Magic to edit the cell). The cell’s contents are stored in a file with this name plus a **.mag** extension. The bottom name inside each bounding box is called an *instance identifier*, and is used to distinguish different instances of the same subcell. Instance id’s are used for routing and circuit extraction, and are discussed in Section 6.

Subcells can be manipulated using the same selection mechanism that you learned in Tutorial #2. To select a subcell, place the cursor over the subcell and type **f** (“**f**ind cell”), which is a macro for **:select cell**. You can also select a cell by typing **s** when the cursor is over a location where there’s no paint; **f** is probably more convenient, particularly for cells that are completely covered with paint. When you select a cell the box will be set to the cell’s bounding box, the cell’s name will be highlighted, and a message will be printed on the text display. All the selection operations (**:move**, **:copy**, **:delete**, etc.) apply to subcells. Try selecting and moving the top subcell in **tut4a**. You can also select subcells using area selection (the **a** and **A** macros): any unexpanded subcells that intersect the area of the box will be selected.

To see what’s inside a cell instance, you must *expand* it. Select one of the instances of **tut4y**, then type the command

:expand toggle

or invoke the macro **ˆX** which is equivalent. This causes the internals of that instance of **tut4y** to be displayed. If you type **ˆX** again, the instance is unexpanded so you only see a bounding box again. The **:expand toggle** command expands all of the selected cells that are unexpanded, and unexpands all those that are expanded. Type **ˆX** a third time so that **tut4y** is expanded.

As you can see now, **tut4y** contains an array of **tut4x** cells plus some additional paint. In Magic, an array is a special kind of instance containing multiple copies of the same subcell spaced at fixed intervals. Arrays can be one-dimensional or two-dimensional. The whole array is always treated as a single instance: any command that operates on one element of the array also operates on all the other elements simultaneously. The instance identifiers for the elements of the array are the same except for an index. Now select one of the elements of the array and expand it. Notice that the entire array is expanded at the same time.

When you have expanded the array, you’ll see that the paint in the top-level cell **tut4a** is displayed more brightly than the paint in the **tut4x** instances. **Tut4a** is called the *edit cell*, because its contents are currently editable. The paint in the edit cell is normally displayed more brightly than other paint to make it clear that you can change it. As long as **tut4a** is the edit cell, you cannot modify the paint in **tut4x**. Try erasing paint from the area of one of the **tut4x** instances: nothing will be changed. Section 4 tells how to switch the edit cell.

Place the cursor over one of the **tut4x** array elements again. At this point, the cursor is actually over three different cells: **tut4x** (an element of an array instance within **tut4y**), **tut4y** (an instance within **tut4a**), and **tut4**. Even the topmost cell in the hierarchy is treated as an instance by Magic. When you press the **s** key to select a cell, Magic initially chooses the smallest instance visible underneath the cursor, **tut4x** in this case. However, if you invoke the **s** macro again (or type **:select**) without moving the cursor, Magic will step through all of the instances under the cursor in order. Try this out. The same is true of the **f** macro and **:select cell**.

When there are many different expanded cells on the screen, you can use the selection commands to select paint from any of them. You can select anything that's visible, regardless of which cell it's in. However, as mentioned above, you can only modify paint in the edit cell. If you use **:move** or **:upsidedown** or similar commands when you've selected information outside the edit cell, the information outside the edit cell is removed from the selection before performing the operation.

There are two additional commands you can use for expanding and unexpanding cells:

:expand
:unexpand

Both of these commands operate on the area underneath the box. The **:expand** command will recursively expand every cell that intersects the box until there are no unexpanded cells left under the box. The **:unexpand** command will unexpand every cell whose area intersects the box but doesn't completely contain it. The macro **x** is equivalent to **:expand**, and **X** is equivalent to **:unexpand**. Try out the various expansion and unexpansion facilities on **tut4a**.

3 Manipulating Subcells

There are a few other commands, in addition to the selection commands already described, that you'll need in order to manipulate subcells. The command

:getcell *name*

will find the file *name***.mag** on disk, read the cell it contains, and create an instance of that cell with its lower-left corner aligned with the lower-left corner of the box. Use the **getcell** command to get an instance of the cell **tut4z**. After the **getcell** command, the new instance is selected so you can move it or copy it or delete it. The **getcell** command recognizes additional arguments that permit the cell to be positioned using labels and/or explicit coordinates. See the *man* page for details.

To turn a normal instance into an array, select the instance and then invoke the **:array** command. It has two forms:

:array *xsize ysize*
:array *xlo xhi ylo yhi*

In the first form, *xsize* indicates how many elements the array should have in the x-direction, and *ysize* indicates how many elements it should have in the y-direction. The spacing between elements is controlled by the box's width (for the x-direction) and height (for the y-direction). By changing the box size, you can space elements so that they overlap, abut, or have gaps between them. The elements are given indices from 0 to *xsize*-1 in the x-direction and from 0 to *ysize*-1 in the y-direction. The second form of the command is identical to the first except that the elements are given indices from *xlo* to *xhi* in the x-direction and from *ylo* to *yhi* in the y-direction. Try making a 4x4 array out of the **tut4z** cell with gaps between the cells.

You can also invoke the **:array** command on an existing array to change the number of elements or spacing. Use a size of 1 for *xsize* or *ysize* in order to get a one-dimensional array. If there are several cells selected, the **:array** command will make each of them into an array of the same size and spacing. It also works on paint and labels: if paint and labels are selected when you invoke **:array**, they will be copied many times over to create the array. Try using the array command to replicate a small strip of paint.

4 Switching the Edit Cell

At any given time, you are editing the definition of a single cell. This definition is called the *edit cell*. You can modify paint and labels in the edit cell, and you can re-arrange its subcells. You may not re-arrange or delete the subcells of any cells other than the edit cell, nor may you modify the paint or labels of any cells except the edit cell. You may, however, copy information from other cells into the edit cell, using the selection commands. To help clarify what is and isn't modifiable, Magic displays the paint of the edit cell in brighter colors than other paint.

When you rearrange subcells of the edit cell, you aren't changing the subcells themselves. All you can do is change the way they are used in the edit cell (location, orientation, etc.). When you delete a subcell, nothing happens to the file containing the subcell; the command merely deletes the instance from the edit cell.

Besides the edit cell, there is one other special cell in Magic. It's called the *root cell* and is the topmost cell in the hierarchy, the one you named when you ran Magic (**tut4a** in this case). As you will see in Tutorial #5, there can actually be several root cells at any given time, one in each window. For now, there is only a single window on the screen, and thus only a single root cell. The window caption at the top of the color display contains the name of the window's root cell and also the name of the edit cell.

Up until now, the root cell and the edit cell have been the same. However, this need not always be the case. You can switch the edit cell to any cell in the hierarchy by selecting an instance of the definition you'd like to edit, and then typing the command

:edit

Use this command to switch the edit cell to one of the **tut4x** instances in **tut4a**. Its paint brightens, while the paint in **tut4a** becomes dim. If you want to edit an element of an array, select the array, place the cursor over the element you'd like to edit, then type **:edit**. The particular element underneath the cursor becomes the edit cell.

When you edit a cell, you are editing the master definition of that cell. This means that if the cell is used in several places in your design, the edits will be reflected in all those places. Try painting and erasing in the **tut4x** cell that you just made the edit cell: the modifications will appear in all of its instances.

There is a second way to change the edit cell. This is the command

:load *name*

The **:load** command loads a new hierarchy into the window underneath the cursor. *Name* is the name of the root cell in the hierarchy. If no *name* is given, a new unnamed cell is loaded and you start editing from scratch. The **:load** command only changes the edit cell if there is not already an edit cell in another window.

5 Subcell Usage Conventions

Overlaps between cells are occasionally useful to share busses and control lines running along the edges. However, overlaps cause the analysis tools to work much harder than they would if there were no overlaps: wherever cells overlap, the tools have to combine the information from the two separate cells. Thus, you shouldn't use overlaps any more than absolutely necessary. For example, suppose you want to create a one-dimensional array of cells that alternates between two cell types, A and B: "ABABABABABAB". One way to do this is first to make an array of A instances with large gaps between them ("A A A A A A"), then make an array of B instances with large gaps between them ("B B B B B B"), and finally place one array on top of the other so that the B's nestle in between the A's. The problem with this approach is that the two arrays overlap almost completely, so Magic will have to go to a lot of extra work to handle the overlaps (in this case, there isn't much overlap of actual paint, but Magic won't know this and will spend a lot of time worrying about it). A better solution is to create a new cell that contains one instance of A and one instance of B, side by side. Then make an array of the new cell. This approach makes it clear to Magic that there isn't any real overlap between the A's and B's.

If you do create overlaps, you should use the overlaps only to connect the two cells together, and not to change their structure. This means that the overlap should not cause transistors to appear, disappear, or change size. The result of overlapping the two subcells should be the same electrically as if you placed the two cells apart and then ran wires to hook parts of one cell to parts of the other. The convention is necessary in order to be able to do hierarchical circuit extraction easily (it makes it possible for each subcell to be circuit-extracted independently).

Three kinds of overlaps are flagged as errors by the design-rule checker. First, you may not overlap polysilicon in one subcell with diffusion in another cell in order to create transistors. Second, you may not overlap transistors or contacts in one cell with different kinds of transistors or contacts in another cell (there are a few exceptions to this rule in some technologies). Third, if contacts from different cells overlap, they must be the same type of contact and must coincide exactly: you may not have partial overlaps. This rule is necessary in order to guarantee that Magic can generate CIF for fabrication.

You will make life a lot easier on yourself (and on Magic) if you spend a bit of time to choose a clean hierarchical structure. In general, the less cell overlap the better. If you use extensive overlaps you'll find that the tools run very slowly and that it's hard to make modifications to the circuit.

6 Instance Identifiers

Instance identifiers are used to distinguish the different subcells within a single parent. The cell definition names cannot be used for this purpose because there could be many instances of a single definition. Magic will create default instance id's for you when you create new instances with the **:get** or **:copy** commands. The default id for an instance will be the name of the definition with a unique integer added on. You can change an id by selecting an instance (which must be a child of the edit cell) and invoking the command

:identify *newid*

where *newid* is the identifier you would like the instance to have. *Newid* must not already be used as an instance identifier of any subcell within the edit cell.

Any node or instance can be described uniquely by listing a path of instance identifiers, starting from the root cell. The standard form of such names is similar to Unix file names. For example, if **id1** is the name of an instance within the root cell, **id2** is an instance within **id1**, and **node** is a node name within **id2**, then **id1/id2/node** can be used unambiguously to refer to the node. When you select a cell, Magic prints out the complete path name of the instance.

Arrays are treated specially. When you use **:identify** to give an array an instance identifier, each element of the array is given the instance identifier you specified, followed by one or two array subscripts enclosed in square brackets, e.g, **id3[2]** or **id4[3][7]**. When the array is one-dimensional, there is a single subscript; when it is two-dimensional, the first subscript is for the y-dimension and the second for the x-dimension.

7 Writing and Flushing Cells

When you make changes to your circuit in Magic, there is no immediate effect on the disk files that hold the cells. You must explicitly save each cell that has changed, using either the **:save** command or the **:writeall** command. Magic keeps track of the cells that have changed since the last time they were saved on disk. If you try to leave Magic without saving all the cells that have changed, the system will warn you and give you a chance to return to Magic to save them. Magic never flushes cells behind your back, and never throws away definitions that it has read in. Thus, if you edit a cell and then use **:load** to edit another cell, the first cell is still saved in Magic even though it doesn't appear anywhere on the screen. If you then invoke **:load** a second time to go back to the first cell, you'll get the edited copy.

If you decide that you'd really like to discard the edits you've made to a cell and recover the old version, there are two ways you can do it. The first way is using the **flush** option in **:writeall**. The second way is to use the command

:flush [*cellname*]

If no *cellname* is given, then the edit cell is flushed. Otherwise, the cell named *cellname* is flushed. The **:flush** command will expunge Magic's internal copy of the cell and replace it with the disk copy.

When you are editing large chips, Magic may claim that cells have changed even though you haven't modified them. Whenever you modify a cell, Magic makes changes in the parents of the cell, and their parents, and so on up to the root of the hierarchy. These changes record new design-rule violations, as well as timestamp and bounding box information used by Magic to keep track of design changes and enable fast cell read-in. Thus, whenever you change one cell you'll generally need to write out new copies of its parents and grandparents. If you don't write out the parents, or if you edit a child "out of context" (by itself, without the parents loaded), then you'll incur extra overhead the next time you try to edit the parents. "Timestamp mismatch" warnings are printed when you've edited cells out of context and then later go back and read in the cell as part of its parent. These aren't serious problems; they just mean that Magic is doing extra work to update information in the parent to reflect the child's new state.

8 Search Paths

When many people are working on a large design, the design will probably be more manageable if different pieces of it can be located in different directories of the file system. Magic provides a simple mechanism for managing designs spread over several directories. The system maintains a *search path* that tells which directories to search when trying to read in cells. By default, the search path is ".", which means that Magic looks only in the working directory. You can change the path using the command

:path [*searchpath*]

where *searchpath* is the new path that Magic should use. *Searchpath* consists of a list of directories separated by colons. For example, the path ".:˜ouster/x:a/b" means that if Magic is trying to read in a cell named "foo", it will first look for a file named "foo.mag" in the current directory. If it doesn't find the file there, it will look for a file named "˜ouster/x/foo.mag", and if that doesn't exist, then it will try "a/b/foo.mag" last. To find out what the current path is, type **:path** with no arguments. In addition to your path, this command will print out the system cell library path (where Magic looks for cells if it can't find them anywhere in your path), and the system search path (where Magic looks for files like colormaps and technology files if it can't find them in your current directory).

If you're working on a large design, you should use the search path mechanism to spread your layout over several directories. A typical large chip will contain a few hundred cells; if you try to place all of them in the same directory there will just be too many things to manage. For example, place the datapath in one directory, the control unit in another, the instruction buffer in a third, and so on. Try to keep the size of each directory down to a few dozen files. You can place the **:path** command in a **.magic** file in your home directory or the directory you normally run Magic from; this will save you from having to retype it each time you start up (see the Magic man page to find out about **.magic** files). If all you want to do is add another directory onto the end of the search path, you can use the **:addpath** [*directory*] command.

Because there is only a single search path that is used everywhere in Magic, you must be careful not to re-use the same cell name in different portions of the chip. A common problem with large designs is that different designers use the same name for different cells. This works fine as long as the designers are working separately, but when the two pieces of the design are put together using a search path, a single copy of the cell (the one that is found first in the search path) gets used everywhere.

There's another caveat in the use of search paths. Magic looks for system files in ˜cad, but sometimes it is helpful to put Magic's system files elsewhere. If the **CAD_HOME** shell environment variable is set, then Magic uses that as the location of ˜cad instead of the location in the password file. This overrides all uses of ˜cad within magic, including the ˜cad seen in the search paths printed out by **:path**.

9 Additional Commands

This section describes a few additional cell-related commands that you may find useful. One of them is the command

:select save *file*

This command takes the selection and writes it to disk as a new Magic cell in the file *file***.mag**. You can use this command to break up a big file into smaller ones, or to extract pieces from an existing cell.

The command

:dump *cellName* [*labelName*]

does the opposite of **select save**: it copies the contents of cell *cellName* into the edit cell, such that the lower-left corner of label *labelName* is at the lower-left corner of the box. The new material will also be selected. This command is similar in form to the **getcell** command except that it copies the contents of the cell instead of using the cell as a subcell. There are several forms of **dump**; see the *man* page for details.

The main purpose of **dump** is to allow you to create a library of cells representing commonly-used structures such as standard transistor shapes or special contact arrangements. You can then define macros that invoke the **dump** command to place the cells. The result is that a single keystroke is all you need to copy one of them into the edit cell.

As mentioned earlier, Magic normally displays the edit cell in brighter colors than non-edit cells. This helps to distinguish what is editable from what is not, but may make it hard for you to view non-edit paint since it appears paler. If you type the command

:see allSame

you'll turn off this feature: all paint everywhere will be displayed in the bright colors. The word **allSame** must be typed just that way, with one capital letter. If you'd like to restore the different display styles, type the command

:see no allSame

You can also use the **:see** command to selectively disable display of various mask layers in order to make the other ones easier to see. For details, read about **:see** in the Magic man page.