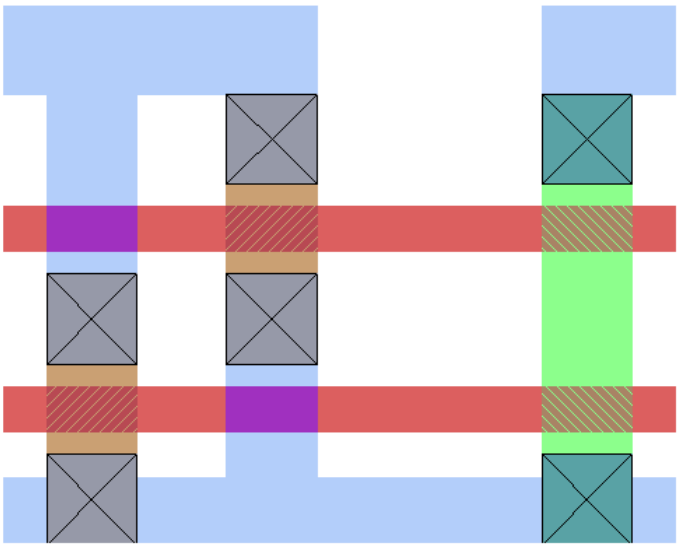
CS/EE 181
Guidelines for Reasonable Layout

(Revised 9 October 1996)



`magic` layout of a NAND gate, a basic building block of CMOS circuitry.

Mika Nyström
Department of Computer Science
CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California, U.S.A.

0. Preliminaries

0.1. Prerequisites

Before you read this, you should have completed the `magic` tutorials according to the instructions of *Homework 1.* You should also be familiar with basic concepts of metal-oxide-semiconductor field-effect transistors (MOSFETs) as explained in Lecture 2.

0.2. Purpose

The purpose of the exercises contained in this handout is to bring together the mechanical knowledge gained from the usual `magic` tutorials and the electrical knowledge gained from Lecture 2. At the end of working through this handout, you should confidently be able to complete homework/lab assignment 1. This handout will also form the basis for the `magic` tutorials arranged for the second week of class.

1. What is layout?

Layout is the process by which a circuit *specification* is converted to a *physical implementation* with enough information to deduce all the relevant physical parameters of the circuit. Circuit layout is not the only thing we do in CS/EE 181, nor is it perhaps the most important thing we do, but it is undoubtedly the most time-consuming.

1.1. The compilation process

From a computer scientist's point of view, the layout process seems familiar enough. We are given a piece of *source code,* this time usually in terms of a circuit diagram, and we want to compile it to an *object code,* the physical layout of the circuit. Where do we find the compiler? Well, for the purposes of CS/EE 181, with some minor exceptions, *you* are the compiler!

2. The role of layout in the design process

The layout step is the last major step in the design process before testing and fabrication; it is the step which reveals to the designer all the subtle electrical characteristics of the hitherto clean and logical digital systems.

2.1. CIF

We communicate with the chip foundry in a file format called CIF (Caltech Intermediate Form—this format is documented in Mead and Conway). In the olden days, chip designers would generate CIF by drawing their designs on graph paper and converting the polygons to CIF by hand. Later on, digitizing tablets were used, but ultimately of course the entire design process was made digital. You will not be required to draw any graph paper layout in CS/EE 181 this year! We will exclusively use `magic` to generate layout, and `magic` does the conversion to CIF for us. (The command to write CIF is simply "`:cif.`")

2.2. Tolerances and design rules

There is a limit to how small features the photolithographic process can generate. Generally this *feature size* is the width of a single minimum-width polysilicon wire used as a transistor gate (since this is the most important physical circuit dimension in determining circuit speed, processes are optimized to generate very short gates). With our tools, the designer generates his designs in terms of a unit which is usually *half* the minimum feature size. Thus, the width of a polysilicon wire is twice the unit of length. Following Mead and Conway, the unit of length is called λ (which in this case is not the wavelength of anything in particular...)

2.3. Scaling

Since physically smaller circuit elements operate faster and at lower power levels than larger ones, there has been a steady push since the birth of integrated circuits to make all the elements of the system as small as possible. Around 1970, it was realized that by scaling *all* parts of a circuit the same amount and also applying a scaling factor to such non-spatial dimensions as doping level and power supply voltage, a circuit with, relatively speaking, identical characteristics to the unscaled one could be obtained. This is the basis of our insistence to measure all our transistors in λs rather than in microns—once you have designed a circuit in a two micron technology, it's perfectly feasible to fabricate the same circuit in 0.35 micron technology and be reasonably confident that the same general circuit elements will work (albeit close to ten times faster and with much less power).

We have said that the smallest feature that we can fabricate on the chip is 2λ wide. However, we can obviously draw a 1λ wire in `magic`... (Try it! What happens?) What does this mean? Well, we are simply trying to specify a wire which is so narrow that it is unmanufacturable—if we submit a design containing such wires to the foundry, many or perhaps all the 1λ wires are going to come back as open circuits, giving us a useless chip. This is the origin of *design rules.*

2.4. SCMOS Design rules

The ability to draw razor-thin wires in `magic` is not the only way in which the designer can attempt to overtax the limits of the fabrication technology. There are design rules to specify many other widths and spacings in order to prevent shorts and opens from occurring. Mutually exclusive layers are also specified in terms of design rules, e.g., there is a rule which prevents nFETs and pFETs from overlapping (as you probably know, this is physically impossible since the substrate can be either p-type or n-type but not both at the same time.)

As everything else on the chip, the design rules also scale with the feature size. That is, the design rules in a modern technology will be similar to those in an old technology with large features *when expressed in* λ*s.*†

† Well, not quite, because modern technologies are often optimized for different kinds of operation from older ones, and this often leads to slightly different design rules. Moreover, newer technologies often offer features not present in the older ones, e.g., three or more metal layers.

A listing of the design rules is available in file `~cad/lib/magic/sys/scmos.tech27`. It is also available in the lab in the form of the report *MOSIS Scalable CMOS Design Rules (revision 7).* This report covers the design rules in terms of CIF layers. For the correspondence between CIF layers and `magic` layers, see the *Magic and CIF Correspondence Table.* The most important design rules are summarized below (all distances are *minimum* design rules).

Polysilicon width	2λ	Polysilicon spacing	2λ	Transistor width	3λ
Metal 1 width	3λ	Metal 1 spacing	3λ	Via size	4λ
Metal 2 width	3λ	Metal 2 spacing	4λ		

Table of selected design rules for 2μm SCMOS process (`magic` design rules).

3. Working with `magic`

A basic feature of VLSI technology (a basic *abstraction barrier*, if you wish) is that the complexity of manufacturing a chip is (almost) entirely independent of the complexity of the structures that the designer has put on the chip.

By now, you realize that designing chips with `magic` involves drawing *layers* of colored paint. These layers somehow correspond to actual physical structures that are fabricated. Although you may for the most part ignore it, the `magic` layers do not correspond directly to *mask* layers which are used directly in the patterning process. If you want to know what is being put in each mask layer, use the `:cif` commands from inside `magic`:

```
:cif see ?
CIF name "?" doesn't exist in style "lambda=1.0(nwell)".
The valid CIF layer names are:  CWN, CMS, CMF, CPG, CAA, CX, CVA, CVA,
CEL, CCE, CCA, CCA, CCP, CBA, XTN, XTP, CSN, CSP, CCD, COG, XP.
```

`magic` frees the designer from having to worry about every detail of which layer has to overlap which layer and by how much and where to put contact cuts, etc., and allows him to concentrate on the most important junctions in the design, the *transistors.**

* One other thing `magic` frees the designer from thinking about is absolute dimensions. `magic` layouts are specified in λs, whereas CIF is an absolute file format. CIF dimensions are specified in *centimicrons.*

3.1. Making a transistor

To design a transistor in `magic`, start by drawing a piece of "diffusion" (i.e., `:paint pdiff` or `:paint ndiff`). This layer does not of course signify that the foundry is going to put silicon there since the entire chip is made of silicon. Rather than that, it means that the indicated area of the chip is made *active,* or in other words that the *field oxide* which insulates the substrate from the devices which are fabricated on top of it is made thinner into a *gate oxide.* Any polysilicon drawn over an active region is turned into a transistor gate. Thus, to make a transistor, cross a piece of polysilicon over a piece of ndiffusion. This makes an nFET. Analogously, a pFET is formed by intersecting the two magic layers `polysilicon` and `pdiffusion`. Try it now!

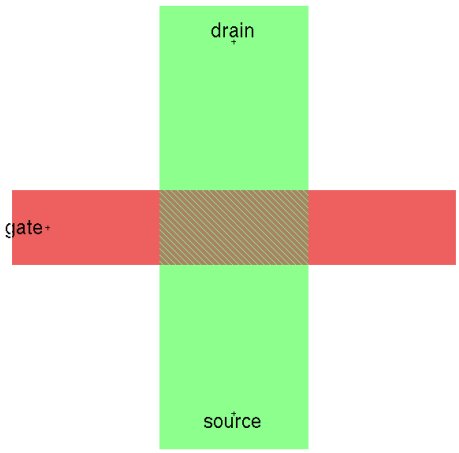


Figure 1. Forming a (nFET) transistor by overlapping polysilicon and ndiffusion. The actual transistor is formed as the intersection of the two layers. The *length* of the transistor is (in this diagram) the height of the intersection and the *width* is the width of the intersection. To reduce confusion, we generally describe the size of transistors in these terms rather than referring to the size of the polysilicon wire (which would result in the opposite nomenclature).

By making the transistor wider, we reduce its "on" resistance and increase its current drive capability. (Think of it as several smaller transistors in parallel.) We call such a transistor *strong.* Conversely, we can make a *weak* transistor by making it longer, thus increasing its resistance and reducing its current drive capability. (Think of it as putting several transistors in series.)

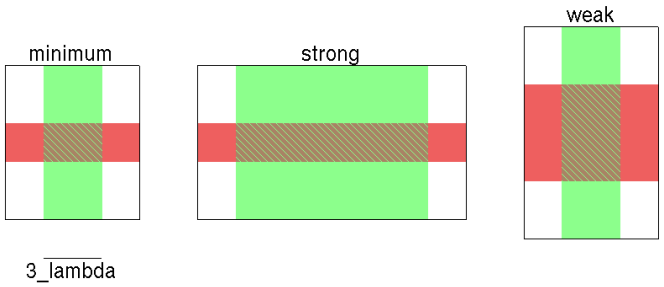


Figure 2. Illustration of minimum size, strong, and weak transistors, respectively. The minimum size transistor is 2λ long and 3λ wide, the strong transistor is 2λ long and 10λ wide, and the weak transistor is 8λ long and 3λ wide. Normally, the smallest transistor that is used is 4λ wide. Weak transistors are used only in special circuits that hold state.

3.2. Using the transistor

Obviously just an unconnected transistor is not very useful. We need to connect the transistor to the outside world. On the actual chip this is done by cutting a hole in the oxide and allowing the `metal1` or `m1` layer (CIF layer `CMF`) to drip onto the active diffusion, forming an ohmic contact. In CIF, this involves drawing four layers, `CSN, CCA, CAA,` and `CMF` (see CIF correspondence table). In `magic`, all you need to do is `:paint ndc` (or `ndcontact`). Since our main routing layer is metal, we need a way to get from polysilicon to metal as well. This is accomplished with a `polycontact` or `pc`. (Note that by default, if you cross two `magic` layers they do not connect; they merely act as two wires insulated from each other. The exceptions to this rule are transistors and contacts.)

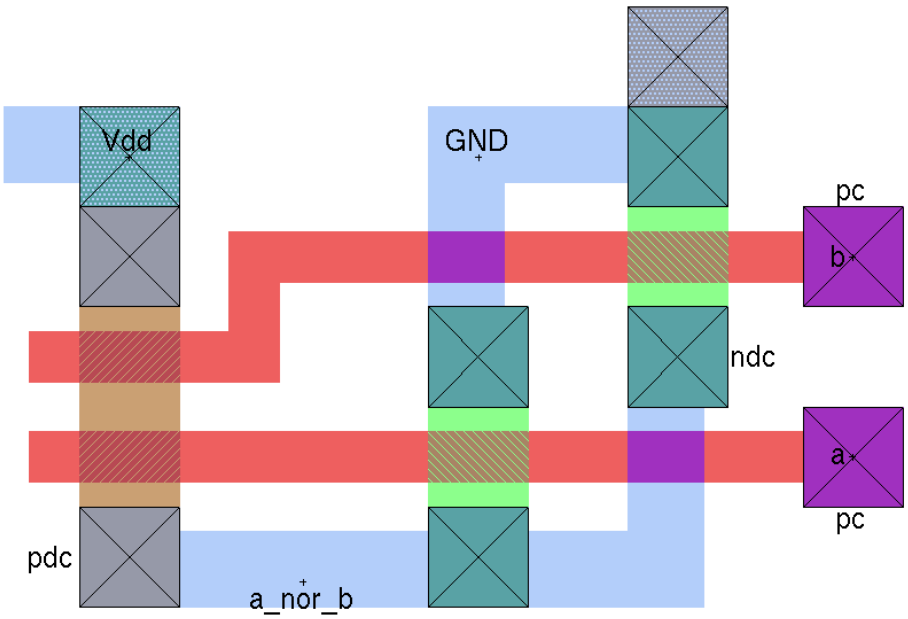


Figure 3. The basic kinds of contacts: `ndc, pdc` and `pc.` Illustrated on a CMOS two-input NOR gate. The pFETs would normally be made wider than in this illustration to make up for the lower mobility of holes which would otherwise reduce the strength of the transistors.

3.3. Longer wires

When we want to run wires farther than just between a few nearby transistors or when we have two busses of `metal1` that cross at right angles, we want to avoid using the rather resistive polysilicon and diffusion layers for signaling. For this purpose, our process offers another layer of metal called `metal2` or `m2`.

3.4. "Plugging" your design

One of the most serious early problems of CMOS VLSI was the phenomenon known as *latchup*. Latchup occurs when parasitic bipolar transistors formed by the sideways p-n junctions on the chip turn on, causing a short-circuit current to flow from the power supply to ground and melting the aluminum wires in the circuit. The electrical details of latchup will be covered later on in class; for now, all you need to know about it is that it can be prevented by adding substrate contacts to your circuit (basically, this means setting the fourth, hidden terminal of the MOSFETs to a known voltage). P-transistors' substrates (i.e., n-wells) are to be tied to `Vdd` and n-transistors' substrates (i.e., p-wells) are to be tied to `GND`. Use the `magic` layers `nwc` and `pwc` to do this. No hard and fast rules exist for how many well contacts are needed—one for every four or five transistors (one for every two transistors with one terminal tied to a power supply) seems to be more than sufficient.

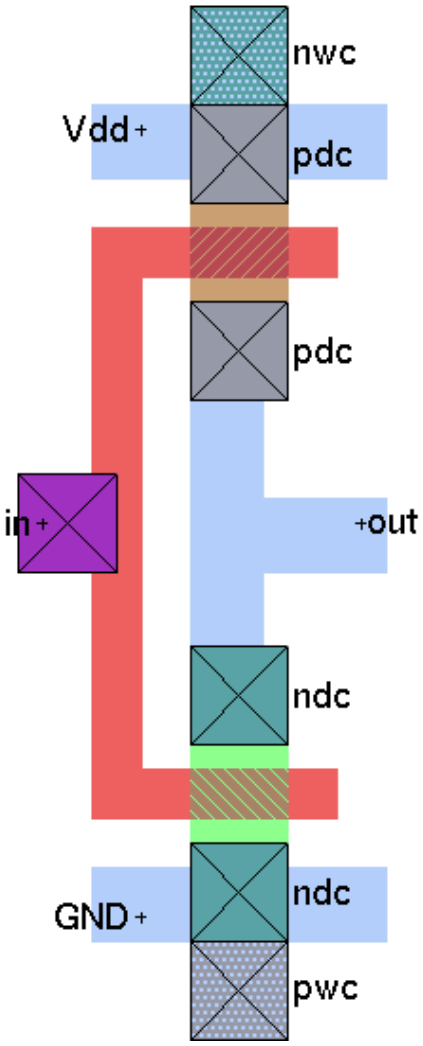


Figure 4. Illustration of the use of well contacts on a CMOS inverter to prevent latchup.

Note that we currently do *not* automatically check for the presence of well contacts. You must do this manually! The TAs will be very careful to check for forgotten well contacts since leaving one out often is a fatal error.

4. How big do I make the transistor?

The first step in learning layout is naturally to be able to design circuits which are topologically correct, and this is all that is required for the first assignment. For later assignments, we shall try to design circuits which are not only topologically correct but that also incorporate some basic electrical engineering knowledge and instinct.

4.1. A CMOS circuit in an electrical engineering nutshell

In a nutshell, a CMOS circuit is a collection of *capacitors* (it is not a coincidence that the symbol for a MOSFET looks much like that for a capacitor) which control current sources and resistive and capacitive interconnections. Since we want to make our circuits as *fast* as possible and since the time constant of a resistor-capacitor network in general is proportional to the product of the resistance and capacitance RC, we want to keep capacitances and resistances as low as possible, or to put it in more appropriate terms, we want to keep capacitances as low as possible and *conductances* as high as possible. Of course, these two goals are often (always) at odds with each other. The goal of making a fast circuit is furthermore often at odds with the goals of reducing *power consumption* and *area*. In short, we have the kind of open-ended design environment which leaves a lot of room for imagination and innovation.

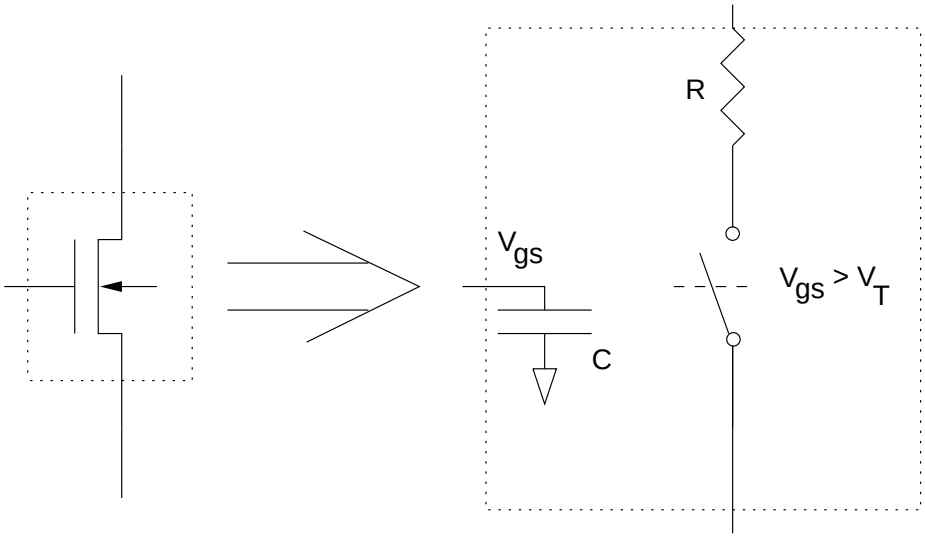


Figure 5. Naïve (but for many purposes, accurate) view of an n-channel MOSFET. The goal of the designer should be, all other things equal, to simultaneously minimize R and C.

Here are some simple guidelines to keep in mind:

1. Keep the transistors minimum-length whenever possible (unless you are designing a weak transistor on purpose, of course). Making transistors longer than minimum length makes them weaker (more resistive) and more capacitive (i.e., they slow down the input signal as well.)
2. Diffusion is rather highly resistive and capacitive, so avoid it as much as possible. In other words, put only 1λ (the minimum) of diffusion between the transistors and the diffusion contacts whenever possible.
3. Because the mobility of holes in silicon is lower than that of electrons, make the p-transistors wider than the n-transistors. A ratio of 2:1 or 1.6:1 in widths works well. However, when you have many n-transistors in series, they are also weak, so in that case, you can make the n-transistors wider as well.
4. Polysilicon is also much more resistive than metal. Avoid it for long interconnections.

More subtly,

5. Metal 2 is run farther from the substrate than metal 1 is. Thus its capacitance per unit area is lower, so use metal 2 for long wires if you can. Especially use metal 2 for long wires with weak drivers (e.g., bus wires that have many drivers connected to them); leave metal 1 for the more powerful control signals. Since data and control wires are usually run at right angles to each other, it will probably be useful to run metal 1 vertically across the design (for control signals) and metal 2 horizontally through the design (for data busses).

Naturally, you are not expected to understand all the subtleties and electrical engineering details that go into these guidelines right away; much of the first term of CS/EE 181 is devoted to exploring these topics. In the end, if you are going to design high-performance systems, expect to spend a great deal of time doing analog simulations to optimize your circuit parameters.

5. Larger designs

There are several important issues to be addressed in order efficiently to design larger systems. Large hardware systems, like large software systems, invite subtle design errors and mismatched interfaces (especially when several people are cooperating on a design). A great deal of care and discipline is required to keep everything straight, and careful planning can easily reduce the time spent designing a large system by a factor of two or more.

5.1. How to partition your design into cells

Partitioning your design into cells is analogous to using *functions* or *subroutines* in software, and how you partition your design should be decided on grounds similar to those used to write a large software system. The basic principle is: *Never design the same cell twice.*

Of course, if your design has many "similar" units in it that are not quite the same, you have a choice. You can just design each kind of unit in a separate cell, or if you prefer, you may design the part that is the same in all the units as a single cell and use other cells to fill in the part or parts that vary. Ultimately, you may want to rethink your design in such a way that the partitioning into "units" makes more sense from a layout perspective.

5.2. Where to start the implementation—floorplanning and system level decisions

How you go about the details of implementing your design is up to you, of course, but a few general hints sometimes help. Start with transistor (circuit) diagrams of the cells you want to implement! Try to choose the roles of the metal layers on the system level, not for each cell. Usually, you will have data traveling "horizontally" and control traveling "vertically" (see above for the usual choice in metal layers). In most designs that you will encounter in CS/EE 181, you can decompose the chip into a control section (usually one more more PLAs) and a datapath section. The very first step of the physical design is referred to as *floorplanning*—this is when you decide which unit goes where and where the datapath goes in relation to the control units

Since the data will be carried on busses across your chip, it becomes *extremely difficult* to change the height of your cells once it is given (all the bits of the adder are right next to the corresponding bits of the register next to it). Therefore, it becomes imperative that you choose the *bit pitch* on the system level. If the bits of the adder are 100λ tall, you will have problems if the bits of the register are not also exactly 100λ tall—at the very least, you will be wasting an inordinate amount of silicon area. If you have a unit that has very complicated cells, make those cells wider, and make the simplest units have very narrow cells.

5.3. Designing your cells

Once you have the transistor diagram for the cells you want to implement, the bit pitch, and the choice of metal directions, you can start wiring up the transistors. If you are going to do analog simulation of your cell, you may want to do that before you wire it up, because it is easier to change when you do not have to move around a lot of metal layers.

Often, in a large cell you often find that some small part of it is difficult to lay out compactly, whereas the rest of it has extra space. A way to take advantage of this is to choose as a good place to start the layout of a complicated cell the area which has the highest wiring density and finish that section first.

Remember that the most restrictive (i.e., "largest") design rule of the SCMOS processes is the spacing between active n- and p-diffusion. It would thus seem like a bad idea to mix nFETs and pFETs haphazardly. Instead, isolate the nFETs in one part of the cell and the pFETs in another and connect them with wires. Although it may lead to longer wiring than if you simply put related transistors next to each other, for the most part it will lead to denser layout.

Finally, remember this is CS/EE 181, not a design competition. It is possible to spend a great deal of time optimizing a single cell in area. It is often better to spend this time on other activities. Make your designs fairly compact, but know when to stop! (This is of course easier if your project is simple enough that you have plenty of space on the chip.)

6. Some layout styles

Each layout artist has his own personal design style, but there are a few "standard" ways of going about things. We briefly examine two popular styles: gate-matrix and towers.

6.1. Gate-matrix layout

Gate-matrix layout has traditionally been very popular in standard synchronous designs and is easily adaptable for automated layout. It is organized and leads to reasonably dense designs.

The basic principle of gate-matrix layout is to run polysilicon in straight wires at right angles to the transistors. By using an 8λ polysilicon grid, contacts can be placed between any two polysilicon wires and (given all the inputs and their inverses) it is quite easy to generate an arbitrary boolean function of the inputs.

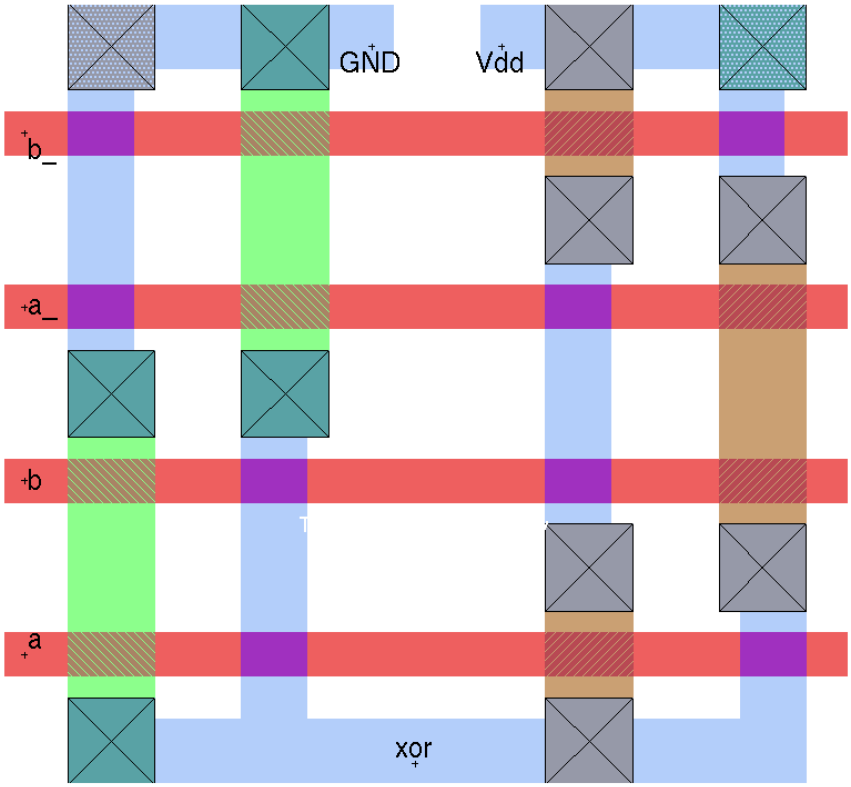


Figure 6. Gate-matrix layout of restoring CMOS two-input XOR gate. Note that diffusion is used to connect series transistors, that the gates are run *straight* and that both senses of the input signals `x` and `x_` need to be provided to the circuit.

6.2. Towers

If you need to design a circuit with large transistors, it is frequently possible to do your layout densely in "towers" of n- and p-diffusion. Remember that you can run metal on top of the transistors on metal1! A trick that is useful to know for this kind of layout is "folded" transistors. By duplicating certain transistors upside down and sharing the output contact between the two transistors, it is possible to design the circuit as unbroken areas of diffusion with wires running between them. You will find that the contacts in the diffusion towers alternate between signal outputs and power supplies. Tower-style layout can be especially useful in modern high-performance layout where large transistors are required.

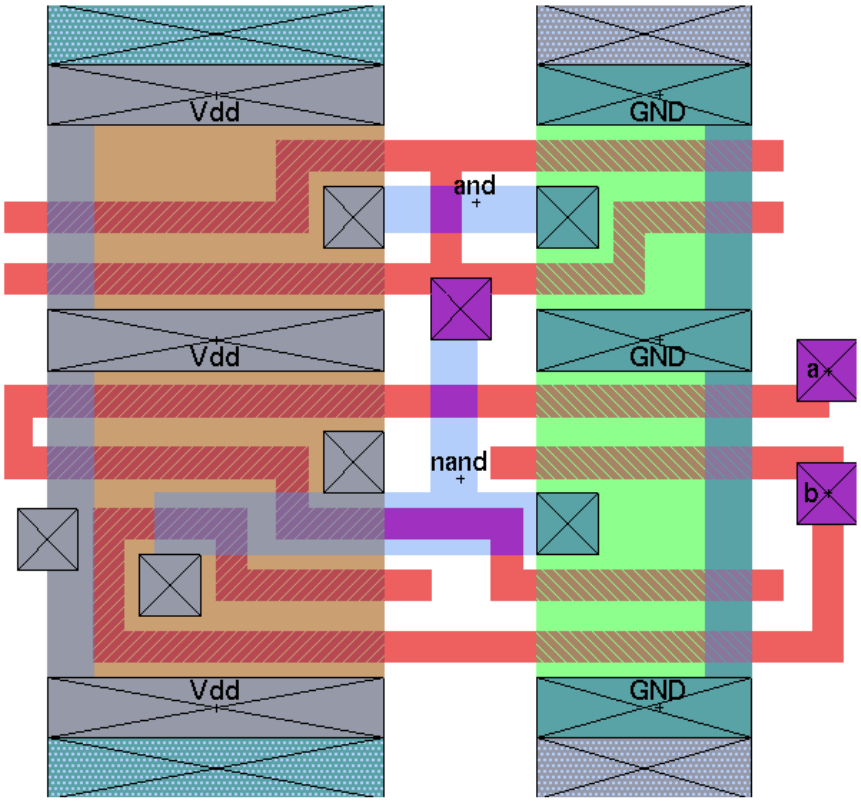


Figure 7. Layout of CMOS two-input AND gate formed as an inverter cascaded after a two-input NAND gate. Note the use of metal1 wires on top of transistors, folded transistors, and contact sharing. Folding a transistor and sharing its output contact allow a reduction of the output capacitance (between e.g., the contact and the substrate of the chip) at the same time as the current drive capability of the transistor is increased. (The gate of a folded transistor is the entire "loop" of polysilicon, and the output is taken from the small drain contact at the center of the loop. The large source contact is connected to a power supply, so the added capacitance here does not slow down the circuit since this node does not switch.) This kind of circuit can operate several times faster than the same circuit designed in gate-matrix style layout.

Finally, there are situations when imposing the extra structure of gate-matrix or tower-oriented layout is more trouble than it is worth. In those situations, feel free to do "random" layout. But be prepared: It may take more time than you expect to get a good layout working.