Homework 6:
Chuck Seitz's Design Competition

Homework is due with relevant files in your CS cluster account or in paper during Friday Rec/OH or to the TA's room at the specified time, NO EXCEPTIONS unless instructor approved. Any late submission will be penalized by weighting errors on those problems that were completed past the due date by the number of days past the due date plus an extra 5% per day. Please indicate which problems, if any, took extra time.

1 Chuck Seitz's Design Competition

70%

2 Brief Problem Statement

This is a notorious *design competition* in which we expect a "good design" as the answer, and may give prizes (sometimes worthwhile, generally worthless) for outstanding designs.

Given two unsigned binary numbers, A and B, your problem is to design a cell of a unilateral iterative array that computes $\min(A, B)$ and $\max(A, B)$. Your cell should compose with itself to perform this computation For any specified number of bits for A and B, e.g., 16 bits, 32 bits, 21 bits, etc. The iterative-array cell that you design must be *static* CMOS combinational logic. The usual rules apply: lows must go through nFETs, highs through pFETs, do not pass any input signals directly to the outputs, and there should be no static power dissipation (i.e., the power supplies should not be connected to each other—even through transistors of the "wrong" kind—except for during the brief instant when inputs are switching).

The judging criteria will be area and thorough verification. The smallest cell design that is also convincingly simulated as operating correctly wins first prize. If we do not regard your simulation output as convincing, your design will be disqualified. Insufficient well plugging, or violation of any design constraints that would not show up in irsim are also grounds for disqualification. Aside from the competition, grading will also be heavily based on these criteria.

See the details below for signal naming conventions and the required signal locations on the boundary of the cell. We recommend that you *read the whole assignment carefully* before you start your design.

3 What to Turn In

1. A cover sheet with your name, the x and y size of your design in λ units, and the area in λ^2 units. The size is determined by the dimensions given by `magic` with the `:box` command when the box surrounds the cell.[1] `magic` defines the size of your cell as the smallest possible box that covers all paint in the cell. It is possible that the horizontal spacing you use when arraying your cell will be less than the width of the box, in which case you should use this spacing for dimensions and the area.
2. A cell table that is labeled with how you have encoded the intercell signals, and a transistor-level circuit diagram of your cell.
3. Leave the following `magic` files in your 181a/lab/lab6 directory:

    `mm.mag` for the cell, and

[1] †

`mm2.mag` for a composition of two cells, labeled as indicated below.

4. Your switch-level simulation (`irsim`) outputs both of a single cell and of a composition of two cells, suitably annotated to convince us that your design functions correctly.

Note: Your composition cell, `mm2,` should contain two instances of `mm` as subcells, not just copies of the paint in `mm.`

4 Design Constraints

1. Your design must be a cell of a unilateral iterative array, with intercell signals propagating left-to-right (MSB to LSB), as described and discussed in the following section.

2. The NORTH side of the cell has the primary inputs, bit i of A and bit i of B, which are available only in positive-logic form. Label these signals within the cell as `a b.` They may enter the north side of the cell on whatever layer and position you wish.

3. The SOUTH side of the cell has the primary outputs, bit i of $\min(A, B)$ and bit i of $\max(A, B)$. Label these signals within the cell as `min` and `max,` respectively. They may leave the south side of the cell on whatever layer and position you wish.

4. The EAST and WEST side of the cell must have signals in positions and layers so that the cell will self-compose east-west. In other words, arraying these cells should connect the corresponding intercell inputs and outputs, and also form the `Vdd` and `GND` connections. The `Vdd` and `GND` signals must be conveyed through the cell on metal: There must be a path entirely on the metal layers between the `Vdd` and `GND` connections on the east of the cell and the corresponding connections on the west of the cell.

5. Just because your cell does not have any geometrical design-rule violations does not assure that the east-west composition of two or more cells will be free of design-rule violations. What if you had p-diffusion near the right edge of your cell, and n-diffusion near the corresponding position on the left edge of your cell? What about metal wires within 1λ of the cell boundaries on both east and west? Be sure to check for design-rule violations or unintended connections when you make the composition cell `mm2.`

6. Label the signals at the boundary of the `mm2` cell `a0 a1 b0 b1 min0 min1 max0 max1,` with the `0` suffix for the LSB and the `1` suffix for the MSB. You will also need to label `Vdd` and `GND,` and connect the left-boundary inputs to appropriate sources. As far as you are concerned, `irsim` only knows about labels that appear in the `mm2` cell, not in the subcells. The safest way for you to attach these labels is to put a small patch of paint in the `mm2` cell on the same layer you are connecting to in the subcell, and label that patch of paint. Although, as part of your verification, you may want to get `irsim` to "watch" the intercell output signals on the right end of the array, please note that producing the comparison result is not part of the problem. Your problem is to produce $\min(A, B)$ and $\max(A, B)$.

7. Use only `poly, ndiff, pdiff, m1, m2` and any necessary contacts between these layers in your layout.

8. Transistors need not adhere to the usual length and width guidlines, they can be the minimum size such that there are no design rule violations, and that circuits simulate correctly.

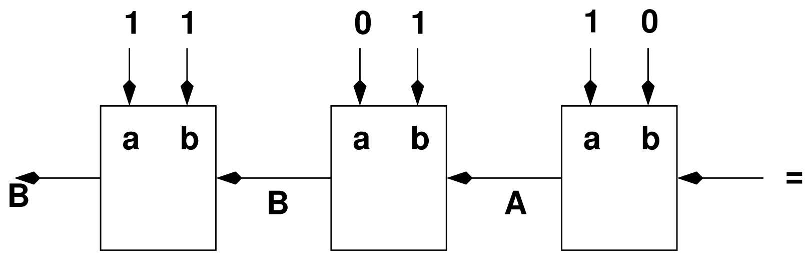
5 Hints and More Details

In order to compute the max and min functions, you must compare A and B. Numerical comparison can be performed with a *unilateral iterative array* with intercell signals propagating from least significant bit (LSB) to most significant bit (MSB)—the usual arithmetic order. (For example, the intercell signal for performing binary addition is the "carry.") Iterative arrays are also the natural way in which to lay out such a function, since you can design a cell for a single bit and then use as many copies as you have bits. The cell is designed to compose with an identical cell on each side. When talking about arithmetic functions, we usually think of composing in the east-west direction, since we write numbers this way. Thus in functions such as addition, the carry propagates right-to-left.

Numerical comparison is unusual in that it can also be accomplished with an iterative array in which the intercell signals propagate left-to-right, that is, from MSB to LSB. Your design is to use this left-to-right comparison.

It happens that, in either case, it is necessary to distinguish three conditions in the intercell signals: "equal so far," "$A > B$," and "$B > A$," which we will abbreviate as "=," "A," and "B," respectively. For left-to-right comparison, the "=" condition should be applied at the left (MSB) boundary, and the comparison result will be produced at the right boundary. For right-to-left comparison, the "=" condition should be applied at the right (LSB) boundary, and the comparison result will be produced at the left boundary.

To check to see if you understand the iterative comparison process, consider comparing the binary numbers $A = 101$, $B = 110$. In a right-to-left comparison:



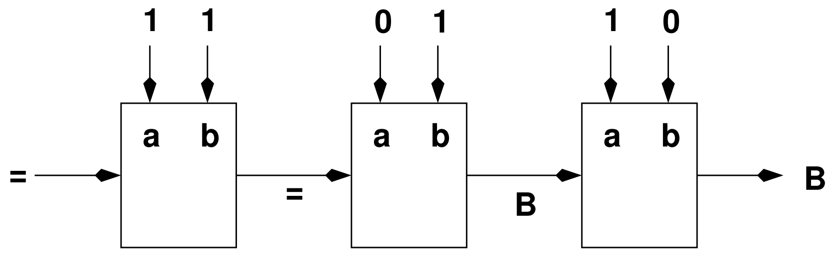
right input \ ab	00	01	11	10
=	=	B	=	A
A	A	B	A	A
B	B	B	B	A

left output

Cell table for RIGHT−TO−LEF comparison

the intercell signal input is less significant (comes from less-significant bits) than the comparison of primary input bits. The required behavior of the right-to-left cell (not the one you will be designing!) is summarized in the *cell table.*

By contrast, with a left-to-right comparison:



the intercell signal input is more significant (comes from more-significant bits) than the comparison of primary input bits. Thus, once it is determined that the numbers are unequal, the decision propagates unchanged through all the remaining cells.

Notice that for left-to-right comparison, the information available at each cell allows the corresponding bit of $\min(A, B)$ and $\max(A, B)$ to be computed from the information available at that cell! If you don't understand this statement, you should stop and figure it out. You must understand this point thoroughly before you go on.

You can then split the design into two parts. The part that performs the comparison is best designed by applying formal methods (see below). The computation of min and max can be designed by seat-of-the-pants methods (just figure out how to steer the data; don't start making truth tables!) as a piece of add-on circuitry for the comparison cell.

The usual formal method for designing iterative array cells is to start with a cell table for the iterative array cell. A cell table is exactly the same idea as the state table for a finite-state machine, but the iteration is in space (by cell number) rather than in time (by time step). After you make an assignment of a distinct binary code for each intercell condition, your cell table becomes a truth table, which can be expected to include "don't care" cases. The three conditions may reasonably be coded on either two or three wires. *The decision of how to code these three conditions on the intercell signals determines most of the essential features of the rest of your design, including its area.* Thus, it may be worthwhile for you to try more than one assignment. The `plamin` program may be useful to you for simplifying the switching functions for different encodings of the three possible conditions of the intercell signals, even though you will not want to use a PLA for the implementation. The minimal switching expressions are a good starting point for figuring out how to implement these functions efficiently in CMOS logic.

6 Some Hints on Producing Compact Layouts

The most important thing to remember is that making a compact layout and making a layout where the routing is easy to follow visually are diametrically opposed goals. Rely on the tools to help follow your wiring rather than relying on a systematic layout pattern.

Draw schematics so that signals enter and exit the schematic on the same side that they will have to enter and exit the layout.

Try to partition the layout. Identify parts of the circuit that have as many (or more) connections to the outside as they have to the inside of the cell, and put these parts near the edges of the schematic. Parts that have many connections to the interior of the circuit should be near the middle of the schematic, and also near the middle of the layout.

Assign layers to signals, based on what each layer does best:

metal2:	wires that go long distances between contacts.
metal1:	wires with several contacts.
poly:	connecting transistor gates.
diff:	connecting series transistors.

Identify the longest wires (e.g., those that go the length of the cell), allocate space for them, and then ignore them until you need them.

Allocate the wider wiring layers in tracks. They aren't very good at bending, jogging, or curving around obstacles. When a track has been assigned for a signal, treat the signal as global (much like clocks and power), easily available from any place along the length of the track.

If a signal has to have a contact, treat the wire as a bus that can be contacted anywhere along its length and used to jumper over other wires. Usually this lets you shorten some of the poly wires (which don't run over the top of circuits the way metal does).

In replicated cells, look for axes of symmetry and points of rotation. Put contacts to common signals (such as inputs, outputs, busses, clocks, power, ground) on the edges (not corners) so they can be shared between cells.

Transistors of the same type whose gates connect to the same signal, or whose source-drain regions connect (especially series transistors) should be kept close together so they can share contacts.

Sometimes reducing the number of transistors in a design can actually make the layout bigger.

Identify the point in the cell with the highest wiring density and begin the layout there. Take advantage of wires that pass all the way through this region by rearranging the circuit elements along them and moving some circuit elements completely out of the crowded region.

Keep your circuit alternatives open while you are doing the layout. For example, if there are two signals that would work equally well as the input to an inverter, write both on the schematic, and don't decide which one to use until you get to that part of the layout.

2 Logical Effort and Combinational Delay Optimization

2.1 Single Path

15%

Optimize the delay for the following multi-stage logic below:

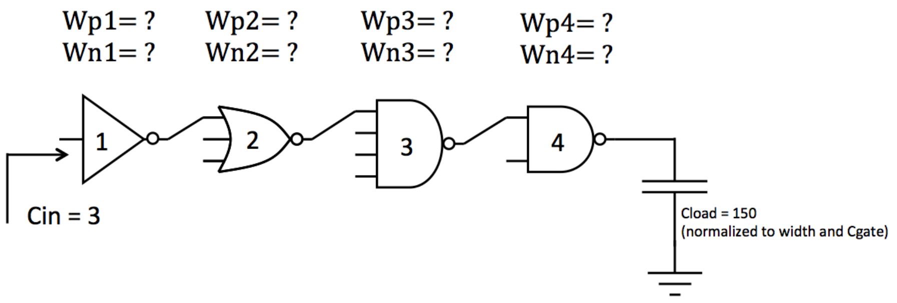


Figure 1: Four stage topology with load.

The Figure 1 above shows a combinational logic path of four elements and a final output load.

a)What should **each** stage's respective PMOS and NMOS widths be such that the overall path delay is minimized?

b)Given that that normalized parasitic capacitance is 1 for an inverter and *n* for any n-input NAND or NOR, what would the total normalized delay for this path be? (Normalized meaning that it is normalized to the delay of a unit inverter). How many F04 delays is that?

You may assume that C_load is the only load this path is required to drive, and that the inverter (input) is minimum sized.

2.2 Branching Path

15%

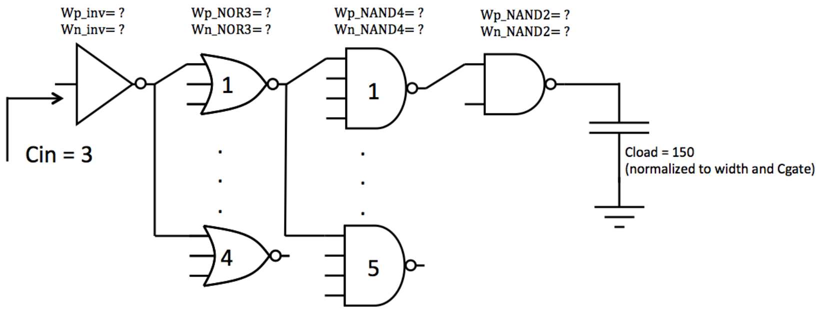


Figure 2: Same four stages, but with branching.

The Figure 2 above shows the same path, but now with branching. Repeat part a) and part b) for this topology.