Homework 2:
Simple Combinational Logic

Homework is due in class on the date indicated above. Late Homeworks 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. (Get started early)

By the end of this assignment, you should

- Have a working knowledge of pass-gate and restoring combinational logic design.
- Understand the concept of hierarchical design.
- Have experience using `alvs`.

1 The ring oscillator

20%

This problem compares the pass-gate and restoring combinational logic using the ring oscillator as example.

i. Before designing the ring oscillator, you need to investigate the open loop performance for both logic. First, connect six cells in series, inverters for the restoring logic and transmission gates for the pass transistor logic. Both cells are shown below.

Second, tie the inputs together and label outputs of the two chains. Simulate the chains by applying a square wave at the input and watch how the outputs behave. (The period of the wave should be large enough to let the output to stabilize)

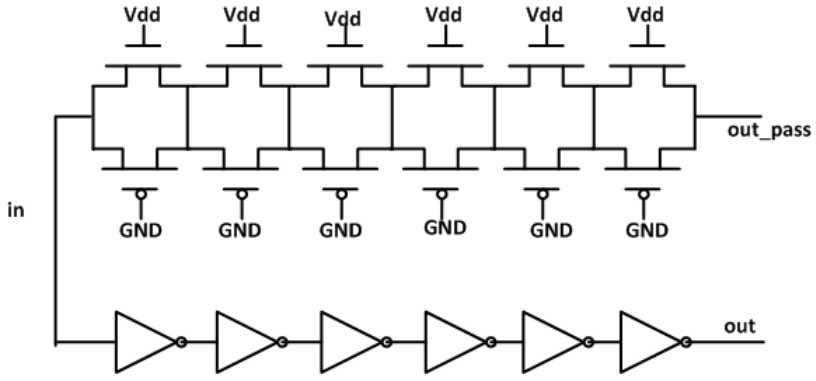

Figure 1: Pass-gate and Restoring Chains.

For the accuracy, `HSPICE` is used for the simulation. `HSPICE` netlist can be generated from the extracted view using magic command "`:ext2spice`". The generated *FILENAME.spice* file describes the connections of different CMOS, their sizes and parasitic capacitance. There is a small problem with this file which can be fixed by `hscript`.

```
% /cs/courses/cs181/hscript FILENAME.spice
```

Include the modified netlist (In this example, assume netlist file is called *inv.spice* and simulation file is called *inv.sp*) file into a simulation file.

One sample `HSPICE` simulation file is given below.

```
****hspice sample code testing inverter's rise and fall time****
.include "inv.spice"
.lib "/cs/courses/cs181/model/PTM_22nm_Metal_Gate_model" cmos_models
.option post
.param sup=1V

vd vdd gnd 'sup'
v0 in gnd pwl 0 'sup' 5n 'sup' 5.001n 0 10n 0 10.001n 'sup'

.tran 10p 20n

.measure rise_time
+ trig v(in)  val='sup * 0.5' fall=1
+ targ v(out) val='sup * 0.5' rise=1

.measure fall_time
+ trig v(in)  val='sup * 0.5' rise=1
+ targ v(out) val='sup * 0.5' fall=1

.end
```

The first line must be left blank or you can write whatever comments. Comments are preceded with "*" in HSPICE files.

The library defines all the CMOS's technical parameters. For this project, 22nm technology library is used.

`.option post` instructs `HSPICE` to write an output file ending in .tr0 containing the simulation waveforms.

`Vd` and `V0` defines two voltage sources. First argument defines the positive node while the second argument defines the negative node. pwl is short for piece-wise linear. All the numbers after pwl come in pairs. The first number defines the time, the second number defines the value of the signal at the time.

`.tran` instructs HSPICE to do transient analysis. The first argument refers to the time step. The second argument is the total simulation time.

`.measure` statements measure rise and fall time. `trig` is the starting point while `targ` is the ending point of measurement.In this example, *rise_time* measures the time from the falling edge of the input to the rising edge of the output. It starts when input drops to 50% of sup, that is, 0.5V. And it ends when output rises to 0.5V.

The "*inv.spice*" is shown below,

```
m0 out in vdd vdd pfet w=88n l=22n
m1 out in gnd gnd nfet w=44n l=22n
```

m0 and m1 defines one PMOS, one NMOS and their sizes. CMOS format is

```
Mx drain gate source body nfet/pfet w=? l=?
```

PMOS is made twice large in order to balance the driving strength of PMOS and NMOS.

For the detail explanation of statements in `HSPICE`, search online for `HSPICE` manual.

Once the file is ready, simulate it using hspice,

```
% hspice inv.sp > log
```

Some files will be generated. The *inv.tr0* file contains the transient behavior of the waveform. The *inv.mt0* file contains the measurement result. *log* contains information about the simulation. Any error or warning will show in this file. You can view the waveform in `cscope`.

```
% cscope
```

In the `cscope`, open the tr0 file and you can play with the waves.

For this part, compare and describe the difference between output waveforms for both chains. Submit the plots of the output waveforms.

ii. Now, close the loop and study the behavior of the ring oscillators. Add one NOR gate to each chain. One of the NOR gate's input is connected to an external *reset* signal. When the *reset* is asserted, the output of the NOR gate is forced to be low. All the other nodes will be forced into one logic value accordingly. When the *reset* is deasserted, the ring oscillator should start the oscillation.

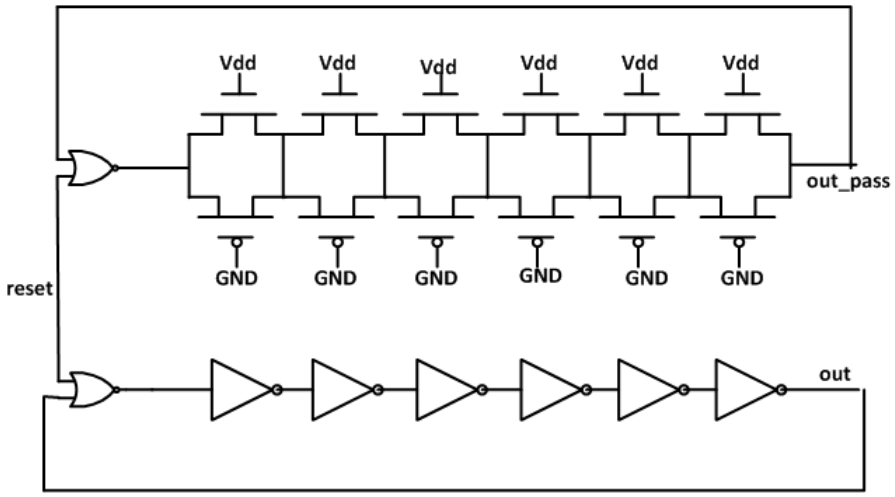

Figure 2: Pass-gate and Restoring Rings.

Follow the same procedure in previous part. Do the ring oscillators oscillate? If yes, what is the period of the oscillation? If no, explain why. Although there is no output, you can label any stage's output as output and watch the waveform at that node. Submit the plots of the waveform at that node.

2 The tricky XOR

20%

This problem is concerned with converting sum-of-products to pass-gate logic.

i. Produce the transistor diagram for a fully restoring XOR gate. From now on, you may use three-terminal transistors in your diagrams, but still plug the wells in the layout. You may assume that you have both senses of the inputs x and y available—it is convenient to call the inverse inputs $x\_$ and $y\_$. How many transistors does the restoring XOR require?

ii. *The Tricky XOR.* The "standard" pass-gate transformation replaces a series transistor with the inverse of that transistor's gate signal:

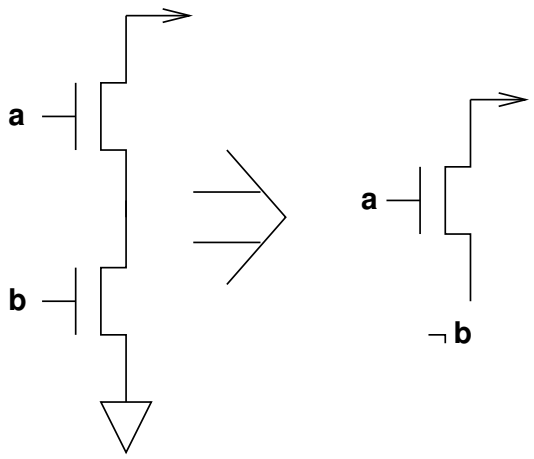

Figure 3: Standard pass-gate transformation replaces a series transistor with a passed-in signal from a previous stage.

Show how to reduce the number of transistors in the XOR gate by half by applying this transformation. Recall the problem that threshold voltages cause when you try to use the wrong kind of transistor: verify that for any permutation of inputs, either there exists a path to a high value that consists only of pFETs, or there exists a path to a low value that consists only of nFETs.

iii. A rudimentary but effective performance metric for a particular logic cell is the maximum number of transistors in series between the power supplies and the output. Assuming in each case that the XOR cell is driven by a circuit that has two pFETs in series and two nFETs in series to its output, how many transistors in series are there between the power rails and the outputs of the restoring version of the XOR? How many for the non-restoring version? What would happen if you were to cascade a large number of pass-gate XORs (i.e., connect the output of one to the inputs of the next)? How would the drive strength (the output conductance) of the circuit compare to the drive strength of a similar cascade of fully restoring XORs?

iv. In many situations, the inputs to a cell are only available in one sense—for the sake of argument, assume that they are available only in the uncomplemented form. In this case, the two XOR designs need to be augmented with two inverters to generate the complemented forms that they require. However, we can use these inverters to our advantage. Since they restore the quality of the signal, we can use them to turn a "tricky" XOR into a "very tricky" XOR that uses the pass-gate XOR internally but does not actually pass any of its inputs to the output (or to each other, even briefly). Show how to design an XOR gate that takes two inputs (uncomplemented inputs only) and generates a restored XOR in nine transistors (including the inverters).

Note. It is possible to generate an XOR design that restores its inputs in only eight transistors, but you will probably find that the two inputs can (for a short time) be connected to each other while one of them is switching, something that could lead one input to switch the value of the other. It would be best if you could find a restoring XOR design that has its input signals going only to transistor gates (i.e., not to pass gates). Of course, if you can figure out both designs, you are welcome to show off your knowledge and turn them both in.

v. Generate `magic` layout for an XOR gate that restores all its inputs. If you managed to figure out the nine-transistor version, feel free to turn that in. Label the inputs `a` and `b` and label the output `xor`. Also label the power supplies `GND` and `Vdd,` as usual. Verify its correct operation as in Lab 1 (using `irsim`), and call the cell `~/cs181/181a/lab/lab2/xor.mag` Leave the `xor.mag` and `xor.sim` files in the directory when you are done.

3 The F-block

30%

In this problem, we will study a way of efficiently generating all boolean functions of n input variables. You will be introduced to hierarchical layout, CAST, and `alvs`.

i. Imagine that we wish to completely specify a k-ary function over the set of k-ary digits $S = \{0, \ldots, k-1\}$ with n parameters, i.e., a function that maps $S^n \rightarrow S$. We can do this by enumerating every possible input in a table, and assigning each row a single output value. This table will have k^n rows. How many distinct such tables are there? (For the special case $k = 2$, we usually call this table the *truth table* of the boolean function.)

ii. There are 16 boolean functions of two inputs. You will be constructing both a fully restoring and a non-restoring *function block* implementing all such functions. Each circuit has eight inputs labeled `a, a_, b, b_`, and either `g0, g1, g2,` and `g3`, or `g0_, g1_, g2_,` and `g3_`, whichever is convenient. It further has one output labeled `f.` The inputs `a` and `b` are the inputs to the boolean function, and `a_` and `b_` their inverses. The inputs `g0`–`g3` or their complement define the function and correspond to the outputs if the inputs are `00, 01, 10,` or `11` respectively. For example (`g0,g1,g2,g3`)=(`0,0,0,1`) is the function AND and (`g0,g1,g2,g3`)=(`0,1,1,1`) denotes OR.

First, produce a transistor diagram of a fully restoring function block in which the pull-up network is the complement of the pull-down network. Using the complement of the pull-down network is not always the best solution: produce another transistor diagram of a fully restoring function block that is "better" than the complemented version (recall that a basic metric for performance is the maximum number of transistors in series between the output and the power supplies). Do not use pass gates.

Using the "better" function block, create a CAST cell-definition in the file `fcell.cast`. It should define but not instantiate the cell `fcell`, which should have six inputs and include two inverters to generate `a_` and `b_` from `a` and `b`. Similarly to the way we simulated the three-input NAND from Lab 1, we shall simulate the production rules of the `fcell`. Notice that the file `fcell.cast` only contains a definition and not an instantiation. This is done so that the file can be used in a hierarchical design. To test it, we need to instantiate it, so create a file `fcell_test.cast` that instantiates a single `fcell` called `FCell`. Also instantiate nodes in the file `fcell_test.cast` that you pass into `FCell`; use the same names as inside the `fcell` definition. The cast manual can be found on the CS181 homepage.

To create a file suitable for `irsim` input, use `prs2sim`:

```
% prs2sim fcell_test
```

Now use `irsim` to test your function block.

After you have completed and tested your `fcell.cast`, create a matching `fcell.mag`. (CAST and `magic` names should match in the following way: each cell definition should have the same name as the corresponding `.mag` file; and each cell instantiation should have the same name as the corresponding `magic` instantiation. You would use the `:identify` command within `magic` to change the name of your `magic` instantiation; to be on the safe side, please use `:extract all` after you change cell-instantiation names. Refer to Figure 2.)

Your design should be *vertically arrayable* for the next part of this problem. This means that you should run power, ground, and the `g0`–`g3`, or `g0_`–`g3_` inputs on metal vertically across the entire cell, stopping and starting at the same location horizontally. You should take the `a` and `b` inputs on metal from the left side, and produce the output `f` to the right. You should now check the layout against your production rules using `alvs` (lvs stands for "layout-versus-schematic").

To perform the `alvs` checks, create a magic cell `fcell_test.mag` that contains a function block identified (`:identify`) as `FCell`; this should match `fcell_test.cast`. Extract this, run `ext2sim`, then run `alvs`. Just like for `fcell_test.cast`, bring nodes passed into `FCell` to the top level by labeling these nodes in the cell `fcell_test` as well as in `fcell`. For `alvs` to work, you must also label `GND` and `Vdd` in `fcell_test`. Note: when labelling, make sure your labels are attached to paint in the cell that they are in. This has several

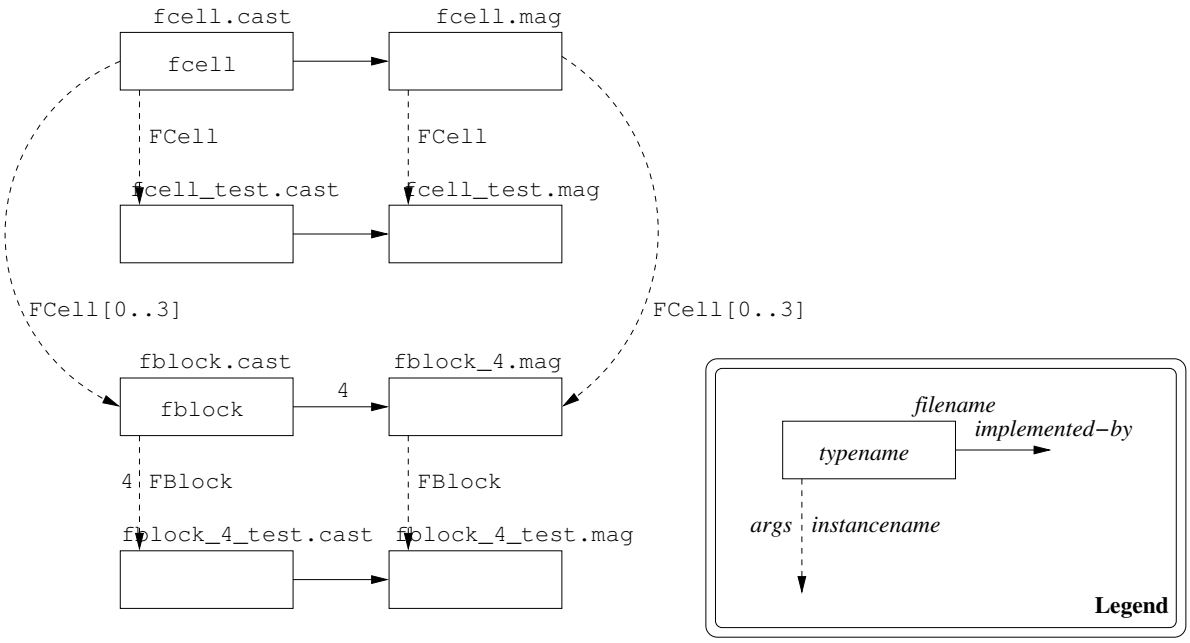

Figure 4: The correspondence between `magic` files, CAST files, `magic` and CAST instances, and CAST types.

consequences: first, if you want to label your design at the top level but there is no paint at the top level, you will have to add paint there; secondly, you should use 0×0 λ labels, because it is easier to see what such labels are attached to.

To generate production rules for `alvs`, use `cflat`, which flattens a hierarchy of production rules:

```
% cflat -lvs fcell_test.cast | alvs -Ddvv fcell_test.sim
```

You are encouraged to run

```
% cflat -lvs fcell_test.cast
```

just to see what `cflat` does. Also study the `alvs` manual page to find out what the command-line options means.

Important: When using `alvs`, make sure that the file fcell_test.sim is the one generated by ext2sim, not prs2sim. Comparing production rules to themselves is not the goal here.

You will get complaints from `alvs` if it finds anything wrong with your layout; if it says nothing, it found nothing wrong with it. Once the layout matches the production rules, re-run the same test you used on the production rules to verify the layout in `irsim`. (To be honest, this test is more a verification of `alvs` than anything else, but it would also catch some kinds of analog problems that `alvs` does not know about.)

The `alvs` program works like this: it checks that the circuit you designed with `magic` implements the CAST specification. To do this, it matches circuit elements attached to nodes that have the same names in CAST and `magic`. This is the only algorithm for matching nodes that `alvs` knows, and this means that *all nodes in your layout must be labeled* and that the labels must match the CAST. Note that it is all right to have "extra" names. If a node is called `a` in the CAST, it could be called `a` and `b` in the `magic`; similarly, it could be called `a` and `c` in the CAST. But at least one name must match for every node in the production-rule set. If you look at the output of `cflat -lvs`, you will see that nodes have names like `FCell.a`; similarly, the nodes in the extract file have names like `FCell/a`: these names should match under the assumption that dots and forward slashes mean the same thing.

For the fully restoring function block you are to produce the following:

- A transistor-level diagram of a fully restoring function block using a complement (dual) for the pull-up network.
- A "better" transistor-level diagram of a fully restoring function block.
- The `fcell.cast`, `fcell.mag`, `fcell_test.sim` files and a transcript of your final `irsim` session. (`fcell_test.sim` should be generated from `fcell.mag`.)

Now you will design a non-restoring function block. For the non-restoring function block you are to produce the following:

- A transistor-level diagram of a non-restoring function block. This should be simpler (in terms of transistor count) than the restoring version.
- A discussion of the pros and cons of non-restoring logic. Please address at least the following points: composability and layout density (would you need to run power supplies to the `magic` cell of your non-restoring function block?)

You do not need to produce layout for the non-restoring function block.

4 The four-bit F-block

30%

i. Generate an `N`-bit function block in `fblock.cast` by arraying your fully-restoring single-bit design. Call the type `fblock`; in other words, you should have the following in your `fblock.cast` file:

```
define fblock(int N)(...)
{
...
}
```

ii. Now design a four-bit function block using `magic` as `fblock_4.mag`. `fblock_4.mag` should match the definition of `fblock` that you have in `fblock.cast` when it is instantiated as `fblock(4)`. Use cell hierarchy: array the subcells vertically! (I.e., make the single-bit function blocks *subcells* of the top-level four-bit function block cell.)

Now create files called `fblock_4_test.mag` and `fblock_4_test.cast` similarly to how you created the files `fcell_test.mag` and `fcell_test.mag`. As with the `fcell`, here you should simply instantiate the `fblock`s as an array called `FBlock`. Note that there are no `fblock_test` files; you cannot instantiate an `fblock` without binding `N`!

In `fblock_4_test.mag` and `fblock_4_test.cast`, name the inputs `a0`...`a3`, `b0`...`b3`, and either `g0`...`g3` or `g0_`...`g3_`. Further label the outputs `f0` through `f3`. You add these names, in the `magic`, by adding labels at the top level of your design; and in the CAST, by introducing new `node`s with those names that you pass in or alias to the already existing nodes. To simulate this design using `irsim`, note that the input and output labels must be in the *top-level cell.* To ensure that they are connected to the wires in the subcells, paint small patches of material in the top level cell overlapping the material in the one-bit cells.

The file `fblock_4_test.cast` will probably look something like the following:

```
import "fblock.cast";
node a0, a1, a2, a3, ...
fblock(4) FBlock({a0, a1, a2, a3}, ...);
```

(Recall that you can use {...} as an array constructor in CAST.) Or if you prefer to introduce new names for the inputs and outputs of `FBlock`:

```
...
node[4] a,b,...
fblock(4) FBlock(a, b, ...);
node a0, a1, a2, a3, ...;
node a0 = a[0], a1 = a[1], ...;
```

Check that `fblock_4_test.mag` and `fblock_4_test.cast` match with `alvs`. Simulate both the CAST and the `magic` with `irsim`; you may find it helpful to put your tests in a file and use the command within `irsim`; this will let you easily repeat the tests.

You are to produce the following:

- A CAST file `fblock.cast` that defines a cell `fblock` that takes a parameter to define the number of bits in the function block.
- The `fblock_4.mag` (that matches `fblock.cast` instantiated as `fblock(4)`), `fblock_test.sim` files (from layout), and a transcript of your final `irsim` session.

Layout Guidelines

1. Try to keep the design as small as possible without violating any design rules and without spending too much time (see note below).
2. Try to follow the general layout guidelines in the *CS181 Guidelines for Reasonable Layout.*
3. For this assignment, try to equalize the driving strength of the nFETs and pFETs. A width ratio of 1:2 usually works well. nFETs should be at least 4λ wide, and pFets at least 8λ wide.
4. Remember to label `GND` and `Vdd` *at the top level of your design.* Also label at the top level any other nodes you want to access by their simple names in `irsim`.

Time Guidelines

You should now be getting very familiar with the tools, especially `magic`. If it is taking too long to produce the layout (e.g. more that two hours for the `fcell`), make sure you have done the tutorials, practice, and ask the TAs for some expert tips! If it is taking too long to produce the layout because you are obsessing over size, understand that layout follows the law of diminishing marginal returns. Later on in the term there will be a homework on compacting layout, so save your energy for that!