ECE 3450: Digital Electronics
Fall 2009, Lab #1

Dr. Pallav Gupta
pallav.gupta@villanova.edu

I Objectives

The goals of this lab are the following:

1. Develop a small cell library for datapath design. In particular, you will design the following cells:
   - NOT, AND2, NAND2, OR2, and NOR2
   - XOR2 and XNOR2
   - AOI22 and OAI22
   - D flip-flop.
2. Perform schematic and layout entry, simulation, DRC, ERC, and NCC for each cell.

II Courses Forum

The forum is located at http://pandim.ece.villanova.edu/phpbbforum. Please use it to ask questions.

III Submission Instructions

The lab must be submitted via the web at http://pandim.ece.villanova.edu. Instructions on how to submit are on the courses forum. Apply the following information on the Remository form when submitting your lab.

Filename:	`lab1-[first name]-[last name].zip`, *e.g.*, `lab1-john-doe.zip`
Location:	`ECE 3450/LAB1`
Title:	`Lab 1`
Author:	Your name, *e.g.*, `John Doe`

Note: Follow the above convention strictly. Failure to do so will result in a **zero**. We request your lab in this specific format so that the automated scripts at the back-end can run smoothly without breaking. Please adhere to it.

See Section XII on how to name and organize the files that you will need to submit for this lab.

IV Collaboration

You must complete this lab independently; you are not allowed to work in pairs or a group. However, you are welcome to discuss the material with your colleagues.

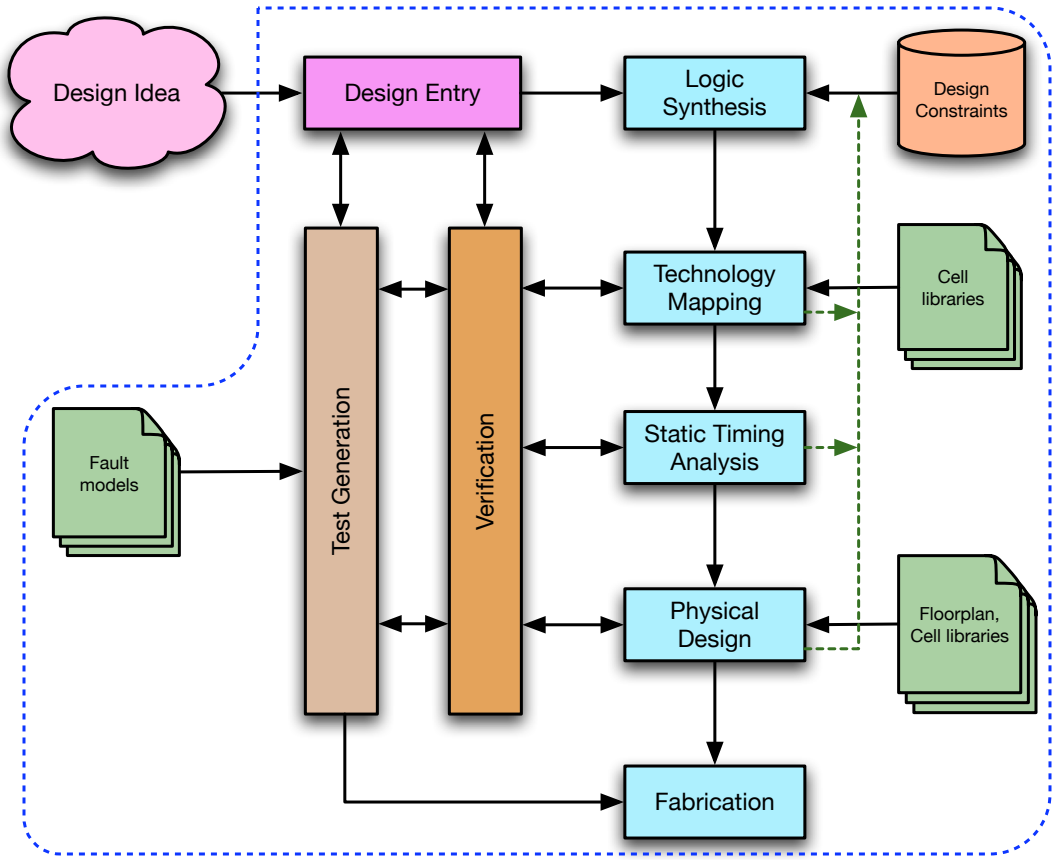


Figure 1: A typical VLSI CAD flow.

V Prerequisites

You must have completed Homework #0 before proceeding further. If not, then it your responsibility to do that first. Otherwise, you will be totally lost in this lab.

VI Introduction

Fig. 1 shows the main components that comprise a typical computer-aided design (CAD) flow for designing VLSI systems. Broadly speaking, there are two types of design styles: *fully-custom* and *cell-based*. In fully-custom design, the designer spends a lot of effort designing individual transistors to extract the maximum available performance. The design cycle is very long and hence, this style is mainly used for microprocessors. In cell-based design (*e.g.*, cell phones, PlayStation, *etc*), the designer opts to use a cell library for circuit layout. In this case, the time-to-market (design cycle) is shorter but performance is sacrificed. Of course, one can mix the two design styles.

We briefly review the basic steps in a typical CAD flow next.

Design Entry: In this initial step, the designer takes a design description, usually specified in a natural language, and converts it into an equivalent form that can be understood by CAD tools. This can be accomplished in a multitude of ways including using programming languages (*e.g.*, C/C++), hardware description languages (*e.g.*, Verilog/VHDL), or schematic capture (*e.g.*, functional or circuit block diagrams). Design constraints, such as minimum clock frequency, total chip area, or power are also identified at this time.

Logic Synthesis: The goal of logic synthesis is to automatically generate an equivalent optimized Boolean circuit from the design specification and constraints. At this point, this circuit may contain high-level Boolean logic primitives (*e.g.*, n-input AND/OR gates, multiplexers, decoders, *etc.*) that can be implemented in several ways. If the design constraints are not satisfied during logic synthesis, then some of the constraints are relaxed or the procedure is repeated using a different yet equivalent design specification.

There are two types of logic synthesis methods: two-level and multi-level. In two-level synthesis, the problem is to find the best circuit that can be implemented using two logic levels (*e.g.*, AND-OR, NAND-NAND, NAND-NOR, *etc.*). Such circuits are fast but not economical. In multi-level synthesis, the problem is to find the best circuit that can be implemented in more than two levels. Such circuits are economical but compromise a little on speed compared to two-level circuits. A mixed approach combines the two methods.

Technology Mapping: Once logic synthesis is complete, each logical primitive in the circuit needs to be mapped onto a logic gate (or a set of logic gates) that is available in the underlying technology. This transformation is called technology mapping. A cell library provides information, such as cell delay, area, and power consumption, about the different cells that are supported by the technology. Once again design constraints must be taken into consideration. For example, a typical constraint might be to ensure that the critical path (*i.e.*, path with the greatest delay) of a circuit contains no more than 30 logic gates.

Static Timing Analysis: After technology mapping, static timing analysis is performed to determine the circuit delay. This must be done to ensure that the circuit performs the required task within the specified time limit (*i.e.*, there are no timing violations). If there are any violations, the circuit must be re-synthesized or re-mapped, and this procedure is repeated. The main goal of static timing analysis is to ensure that when the circuit is fabricated, it can operate at the specified clock frequency.

Physical Design: The goal of physical design is to try to determine the most efficient way to layout and interconnect the gates of a circuit on a chip. In doing so, one of the main objectives is to minimize the total chip area and the amount of interconnect, while ensuring that the circuit meets the timing restrictions. A floorplan aids in this process by providing an initial estimate of the location of the gates. If a circuit cannot be placed and routed without violating the design constraints, it must be re-synthesized or re-mapped. In cell-based design, the cells snap together like LEGO blocks and are interconnected. Design rule check (DRC) and layout-versus-schematic (LVS) (also known as network consistency check (NCC)) check is performed to ensure the chip is manufacturable.

Fabrication: In the final step, a set of masks are generated from the circuit layout and this information is transferred to a semiconductor vendor (*e.g.*, TSMC, MOSIS, *etc*). This process is known as *tapeout*. The chip is then fabricated and tested. If it passes the tests, it is ready for volume production and shipment. Otherwise, the problems must be debugged. If this results in major changes to the design, the entire CAD flow needs to be repeated.

Verification: As can be seen in Fig. 1, verification is done at all phases to find errors in a design. Different methods are used depending upon the level at which verification is being done. In the initial stages, the goal is usually to detect logical errors in the design. As the design matures, the goal shifts towards detecting timing violations. Performing verification early in the design process is relatively much easier and cheaper than doing it during the later stages.

Test Generation: The goal of test generation is to automatically generate a sequence of test vectors that will be applied to a chip, post fabrication, to test for faulty components. These faults can result from a variety of causes, including manufacturing defects, design errors, or in-field operation. Different fault models that target different types of faulty behavior are used in order to generate a comprehensive test set. Test generation is also done early because the test vectors that are generated using fault models developed at the higher level of the design hierarchy, have shown to cover a large proportion of faults/defects that occur during fabrication.

As mentioned earlier, a cell library is a set of primitive digital cells and their characterized models for geometry (area), timing (delay), power, noise, and many more. Commercial cell libraries are available for *all* process technology nodes. However, these libraries are expensive and protected by non-disclosure agreements (NDA). Due to the NDA, they often do not contain the schematic and layout of the cell. In addition, they work on commercial CAD tools which are not used in this course (due to complexity/maintenance of the tools). Here, we

are using Electric, an open-source complementary metal-oxide semiconductor (CMOS), very large scale integrated (VLSI) physical design tool. However, Electric does not provide a cell library. Thus, the goal of this lab is to create a small library, comprising of some datapath cells, that reflects a typical commercial library. You will design and layout the cells. Characterizing them is beyond the scope of this course.

In general, there are two types of cell libraries: *datapath cell* and *standard cell*. Datapath cell libraries are used for datapath design (*e.g.*, the datapath of microprocessors). Significant time and effort is spent on creating highly-optimized cells. Standard cell libraries are used to map designs that are automatically synthesized by CAD tools. It is not uncommon for entire processor cores to be synthesized these days. They differ from datapath cell libraries in that the cell height is usually shorter and the interconnections are made on dedicated wiring channels/tracks. In datapath cell libraries, interconnections are made through (over) the cell.

VII Provided Files

All the files for this lab are provided in `lab1.zip`. Unpack this archive and you will see the following directory structure:

```
lab1/
├─README                                  readme
├─lab1-xx.jelib                          Electric cell library
└─sim/                                   IRSIM test vectors for cells
  ├─ and2.cmd
  ├─ aoi22.cmd
  ├─ dff.cmd
  ├─ nand2.cmd
  ├─ nor2.cmd
  ├─ not.cmd
  ├─ oai22.cmd
  ├─ or2.cmd
  ├─ xnor2.cmd
  └─ xor2.cmd
```

The Electric file is the library containing everything you did in Homework #0. You will add the remaining cells to this library. The `sim/*.cmd` files contain the test vectors to simulate the cell using IRSIM. You should spend a little time browsing through some of these files to get familiar with the syntax. Note that only a couple of the files contain the test vectors. You will need to write your own test vectors for some of the cells.

VIII Designing the Cell Library

Read through the following sections carefully to design the datapath cell library.

VIII.A Before You Begin

The layouts must obey the same constraints as those in Homework #0. In particular, keep the following in mind:

1. Power and ground lines run horizontally in `metal1` on a 90λ center-to-center spacing.
2. All transistors, wires, and well contacts fit between the power and ground lines.
3. All transistors should be within 100λ of a well contact.
4. Avoid long routes in diffusion.
5. You may find it necessary to add a large rectangle of `N-well` or `P-well` to surround the transistors and eliminate spacing problems.
6. All inputs and outputs must be placed on `metal2` contacts and aligned with well contacts.

7. Ensure that all the cells pass DRC, ERC, NCC, and IRSIM simulation.
8. Create neat and clean layouts to avoid problems in the following labs.
9. Export all inputs, outputs, power, and ground.

Important: Before doing any layout, it is critical to draw an accurate **stick diagram** of the cell on paper. Once you are confident, it is then easy to transfer it onto an actual cell layout. I strongly recommend that you follow this practice to save yourself a lot of time (and possible misery).

VIII.B IRSIM Simulation

Please use IRSIM for layout and schematic simulation. See Section IX on how to use IRSIM. It is possible to use ALS but it is more cumbersome. You can use either IRSIM or ALS although I recommend the former.

VIII.C NOT

You have already designed the schematic and icon of the NOT gate in Homework #0. Complete the design by laying out the gate *exactly* as shown in Fig. 2a. Export `a` and `y` as `Input` and `Output`, respectively. Also export `vdd` and `gnd` as `Power` and `Ground`, respectively.

Perform DRC, ERC (well-check), NCC, and IRSIM simulation on the layout. The test vectors are given in `not.cmd`.

VIII.D AND2 and NAND2

You have already completed the design for the NAND2 gate in Homework #0. There is nothing more to be done for this.

Rather than laying out the AND2 gate from scratch, you can save time and effort by employing *hierarchical design*. Recall that an AND2 gate can be built by using NAND2 and NOT.

Create a new cell layout for `and2`. Choose Cell in the `Components` panel and select `nand2{lay}`. Drop it on the layout. Repeat the process with `inv{lay}`. Select both cells by choosing Edit→Selection→Select All. Choose Cell→Expand Cell Instances→All The Way to see what is inside the cells. Move the `nand2` until its cell center is aligned at the top of the `and2`'s cell center. Move the `inv` until its power and ground wires abut those of `nand2`. The arrow keys are useful to perform the movement.

Connect the output of the `nand2` to the input of the `inv` using `metal1`. Also connect the power and ground wires of the two cells using `metal1`. Export inputs `a` and `b` of `nand2` and output `y` of `inv` as the new inputs and output of `and2`. Also export `vdd` and `gnd` as `Power` and `Ground`, respectively.

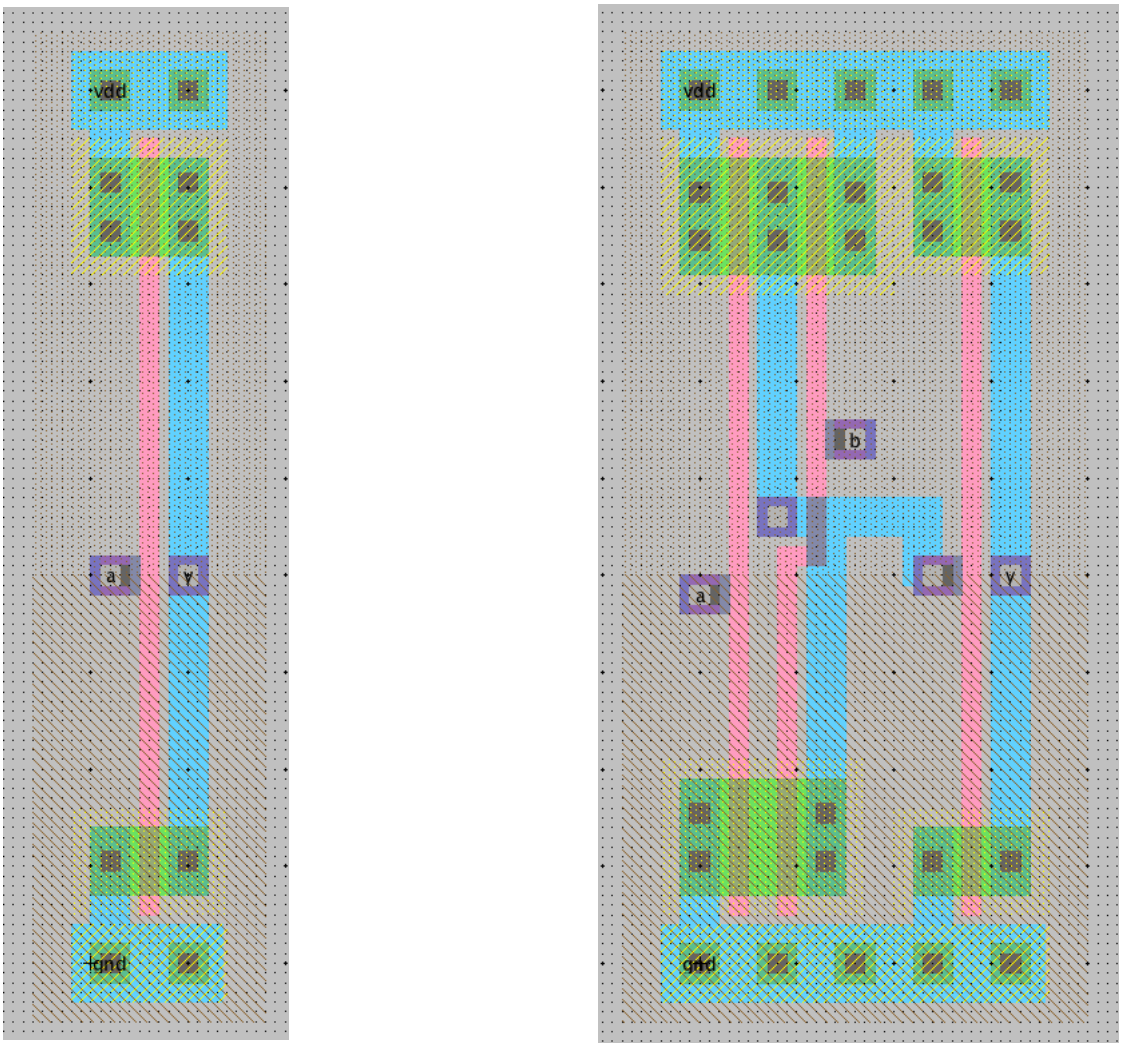
Important: Remember that even though the cells appear to touch (abut), Electric only understands that the power and ground wires should be connected if you explicitly draw an arc between them.

Perform DRC, ERC, NCC, and IRSIM simulation on the layout. The test vectors are given in `and2.cmd`.

VIII.E OR2 and NOR2

The truth tables and icons of the 2-input OR (OR2) and NOR (NOR2) gates are shown in Fig. 3. Based on this, design the following:

NOR2:



(a) NOT.

(b) AND2.

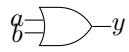
Figure 2: Layout of cells.

- `nor2{sch}`: Use static CMOS. Set the widths of the nMOS and pMOS transistors to 8 and 16, respectively. Export `a`, `b` and `y` as `Input` and `Output`, respectively. Perform DRC and IRSIM simulation on the schematic. The test vectors are given in `nor2.cmd`.
- `nor2{ic}`: Make sure the icon size is 12x6.
- `nor2{lay}`: Export inputs, output, power and ground lines. Perform DRC, ERC, NCC, and IRSIM simulation on the layout.

Once NOR2 has been designed successfully, use hierarchical design to design OR2 from NOR2 and NOT. Perform DRC, ERC, NCC, and IRSIM simulation on the layout. The test vectors are given in `or2.cmd`. As in the case of NOR2, you should have `or2{sch}`, `or2{ic}`, and `or2{lay}` in the library.

VIII.F XOR2 and XNOR2

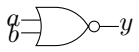
The truth tables and icons of the 2-input XOR (XOR2) and XNOR (XNOR2) gates are shown in Fig. 4. Based on this, the Boolean equations for XOR2 and XNOR2 are $y = \bar{a}b + a\bar{b}$ and $y = \bar{a}\bar{b} + ab$, respectively. If we



(a) OR2

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

(b) OR2.



(c) NOR2.

a	b	y
0	0	1
0	1	0
1	0	0
1	1	0

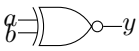
(d) NOR2.

Figure 3: Truth tables and icons for OR2 and NOR2.

(a) XOR2.

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

(b) XOR2.



(c) XNOR2.

a	b	y
0	0	1
0	1	0
1	0	0
1	1	1

(d) XNOR2.

Figure 4: Truth tables and icons for XOR2 and XNOR2.

designed a static CMOS circuit to implement these equations, an extra inverter will be required to invert the output. This is because CMOS is an inverting technology. However, realizing that XOR and XNOR are the complements of each other, we can implement one by using the Boolean equation of the other! That is, we design a static CMOS gate that implements $f = \overline{\bar{a}\bar{b} + ab}$ for XOR. To implement XNOR, we design a static CMOS gate that implements $f = \overline{a\bar{b} + \bar{a}b}$. But if we use this design, the gates are still slow because of a shared diffusion in the pMOS network.

We can improve this design further by recognizing that XOR2/XNOR2 are symmetric functions. Symmetric functions are functions in which permuting the set of variables does not change the original function. This special property allows us to simplify the static CMOS design rule by making the pMOS network identical (rather than complementary) in structure to the nMOS network! This is because at any given instance, only one path through the network will be conducting. Using this idea for the XOR2 gate, the nMOS network must implement $ab + \bar{a}\bar{b}$ while the pMOS network must implement $\bar{a}b + a\bar{b}$. Notice the two equations are complement of each other. Compare this design with the previous equation and you will note that there is no shared diffusion in the pMOS network. The XNOR2 can be designed in a similar fashion.

XOR2:

- `xor2{sch}`: Use static CMOS with identical nMOS/pMOS networks. Set the widths of the nMOS and pMOS transistors to 14 and 20, respectively. Since $\bar{a}$ and $\bar{b}$ are required, you will also need two inverters. Do not assume that the complements are available. Set the width of the nMOS and pMOS transistors in the inverters to 6 and 9, respectively. Export `a`, `b` and `y` as `Input` and `Output`, respectively. Perform DRC and IRSIM simulation on the schematic. You need to create the test vector stimuli in `xor2.cmd`.
- `xor2{ic}`: Make sure the icon size is 12x6.
- `xor2{lay}`: Export inputs, output, power and ground lines. Do not forget to layout the inverters. Perform DRC, ERC, NCC, and IRSIM simulation on the layout.

Repeat the above procedure and design the XNOR2. You will notice that the structure of the XOR2 and XNOR2 are identical (refer to the equations). Therefore, you just need to redo the wirings in this case. Perform DRC, ERC, NCC, and IRSIM simulation on the layout. You need to create the test vector stimuli in `xnor2.cmd`. As in the case of XOR2, you should have `xnor2{sch}`, `xnor2{ic}`, and `xnor2{lay}` in the library.

Important: Do no use hierarchical design on XNOR2 (*i.e.*, XOR2 and NOT) as that requires an extra inverter.

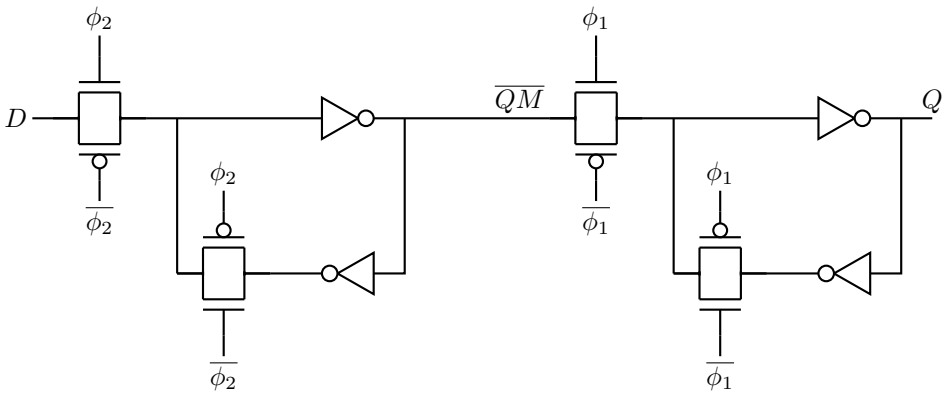


Figure 5: Logic diagram of a D flip-flop.

VIII.G AOI22 and OAI22

The complex gates AND-OR-INVERT (AOI22) and OR-AND-INVERT (OAI22) have four inputs (`a-d`) and an output (`y`). The Boolean equations for AOI22 and OAI22 are $y = \overline{ab + cd}$ and $y = \overline{(a + b)(c + d)}$, respectively. Use static CMOS to implement the two gates. Set the widths of the nMOS and pMOS transistors to 14 and 20, respectively. Export `a-d` and `y` as `Input` and `Output`, respectively. Also export the power and ground lines. Perform DRC and IRSIM simulation on the schematic. Make appropriate icons for the gates. Perform DRC, ERC, NCC, and IRSIM simulation on the layouts. You need to create the test vector stimuli in `aoi22.cmd` and `oai22.cmd`. You should have the following in the library: `aoi22{sch}`, `aoi22{ic}`, `aoi22{lay}`, `oai22{sch}`, `oai22{ic}`, `oai22{lay}`.

VIII.H D Flip-Flop

The D flip-flop is a sequential element with the characteristic equation, $Q = D$ (alternatively, $Q^+ = D \cdot CLK + \overline{CLK} \cdot Q$). When the CLK rises, D is copied into Q. Otherwise, the flip-flop maintains it state. As we have studied in class, clock skew can lead to race conditions. However, this can be alleviated if two non-overlapping clocks, ϕ_1 and ϕ_2 are used. As long as the non-overlap exceeds the clock skew, the flip-flop will function correctly.

The logic diagram of a D flip-flop is shown in Fig. 5. As can be seen, it contains master and slave latches. The slave latch is controlled by ϕ_1, while the master latch is controlled by ϕ_2. You need to layout and test the D flip-flop. Since the design is relatively complex, you are being provided with the schematic (`dff{sch}`) and icon (`dff{ic}`) of the D flip-flop. The size of all the transistors are specified on the schematic which is shown in Fig. 6. Note that inverters have been added at the input and output for signal restoration (in case a weak signal comes in at the input). Also note that the signals `ph1`, `ph1b`, `ph2`, and `ph2b` correspond to ϕ_1, $\overline{\phi_1}$, ϕ_2, and $\overline{\phi_2}$, respectively. Study the schematic until you are confident of its operation.

It is your job to design the layout from this schematic. Design a D-latch and then interconnect two of them. Export the necessary inputs, outputs, clocks, power, and ground lines. Perform DRC, ERC, and NCC on the layout. Simulate the layout with IRSIM to ensure correct operation. The test vectors are given in `dff.cmd`.

IX Using IRSIM

IRSIM comes as a (separate) plugin to Electric. If you wish to use it to simulate your designs, the simplest way to invoke it from the command line is as follows:

```
java -classpath electricBinary-[version].jar:electricIRSIM-[version].jar
                com.sun.electric.Launcher
```

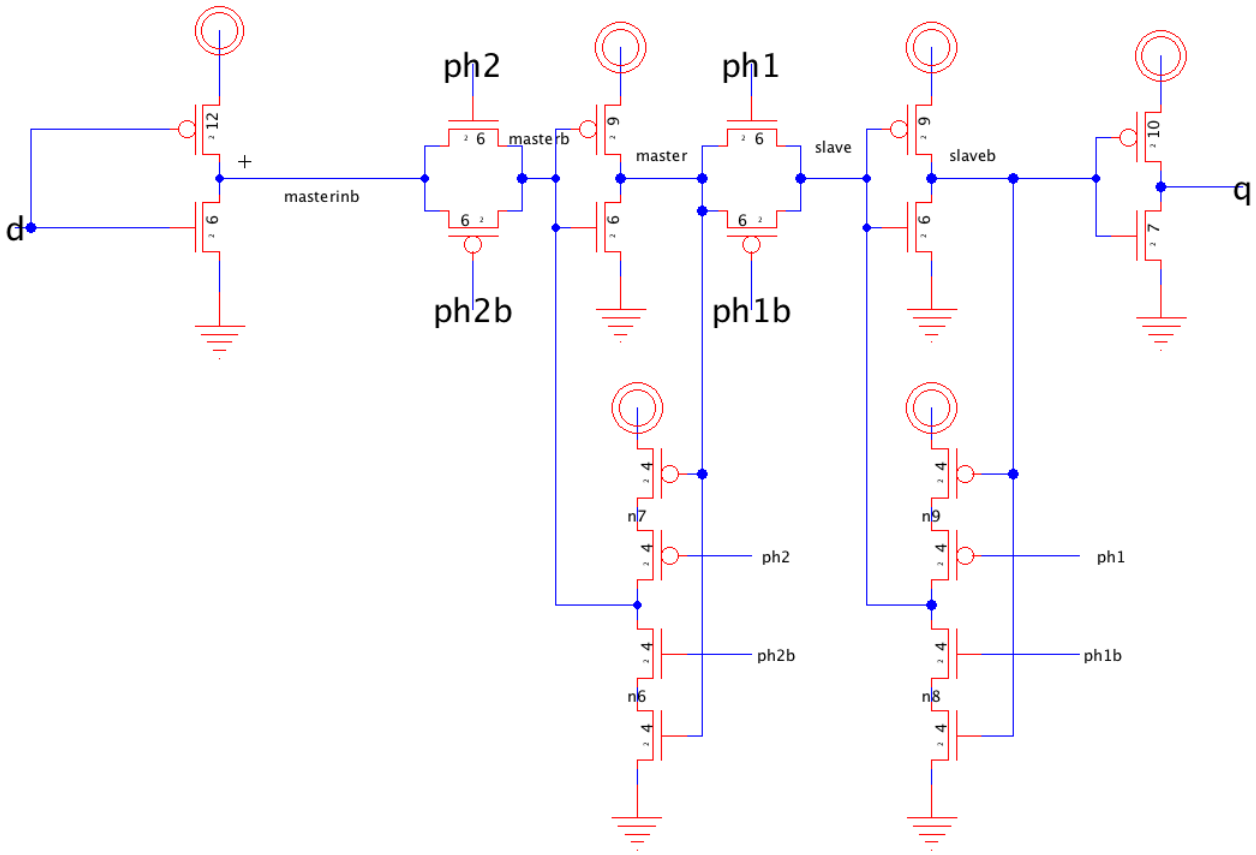


Figure 6: Schematic diagram of a D flip-flop.

On Windows, you must use the semicolon to separate the jar files and you might have to quote the collection. Therefore, try

```
java -classpath "electric-[version].jar;electricIRSIM-[version].jar"
com.sun.electric.Launcher
```

Take advantage of IRSIM command files in the simulation. Sample `*.cmd` files are given in the `sim/` directory. They include test vectors for the input combinations. Open some of the command files with a text editor and browse through them. The syntax of the command file is shown in Table 1. See the IRSIM manual for a complete list of available commands. Assertions are very useful for larger designs because they allow automatic testing of the design without inspecting the waveforms for correctness. Signals retain the last value they were assigned if not explicitly changed. You should use assertions wherever possible for quick debugging.

Invoke IRSIM by choosing Tools→Simulation (Built-in)→IRSIM: Simulate Current Cell. The waveform viewer should open up. Read in the vectors by choosing Tools→Simulation (Built-in)→Restore Stimuli from Disk. Choose the appropriate file and perform the simulation. Check the waveforms to verify that you obtain the correct outputs.

X Technical Report

A technical report (not to exceed 10 pages) is to be written that details everything you did in this lab. You should present the designs of the cells, and show the schematics and layouts. You may also include stick diagrams for your layouts. Furthermore, you should show the results of the simulations, and whether the layouts passed

Table 1: IRSIM Commands

Command	Description
h `sig`	Set `sig` high (logic 1)
l `sig`	Set `sig` low (logic 0)
s `[time]`	Simulate for `[time]` nanoseconds (otherwise use default step)
assert `sig val`	Check that `sig` has the value `val`; emit warning if it does not
x `sig`	Set `sig` to invalid level

DRC, ERC, and NCC or not. Elaborate on any difficulties faced and the employed workarounds. Summarize what you have learned in this lab.

The technical report must be of the highest standards. Otherwise, it runs a high risk of being rejected which will impact your lab grade. You can consult some technical publications to see how to write a good technical report. It must be written using the LaTeX template that was provided earlier. The template can be downloaded at http://pandim.ece.villanova.edu.

XI Parting Words

Congratulations on finishing this lab! Hopefully, you now know how to design, layout, and simulate cells. You should also be comfortable in using Electric by now.

XII What to Submit

For this lab, you must submit the following files:

1. The Electric cell library. Name it `lab1-xx.jelib` where `xx` are your initials.
2. The LaTeX PDF file which contains the technical report. Name it `report-xx.pdf` where `xx` are your initials.

Take both the files and archive (zip) them into a folder. Name the folder `lab1-[first name]-[last name].zip`. See Section III on how to submit the archive. Failure to follow these instructions will result in a **zero** for the lab. No ifs, buts, *etc.*

Important: The Electric library must contain the schematics, icons, and layouts for NOT, AND2, NAND2, OR2, NOR2, XOR2, XNOR2, AOI22, OAI22, and D flip-flop. All schematics must pass DRC and IRSIM simulation. The layouts must pass DRC, ERC, NCC, IRSIM simulation, and be drawn as per specification (outlined above). Icons must be of the correct size.

XIII Errors

I usually write precise tutorials and bug-free code. However, I am human (do not be surprised) and do make mistakes. In addition, CAD tools get updated frequently and the interface might change, rendering parts of the writeup ineffective. If you find any mistakes or inconsistencies while doing this lab, please bring it to my attention **immediately**. You will earn extra points if what you report is indeed a bug.