# CS/EE 181a 2010/11 Lecture 5

General topic of today's lecture: *Time.*

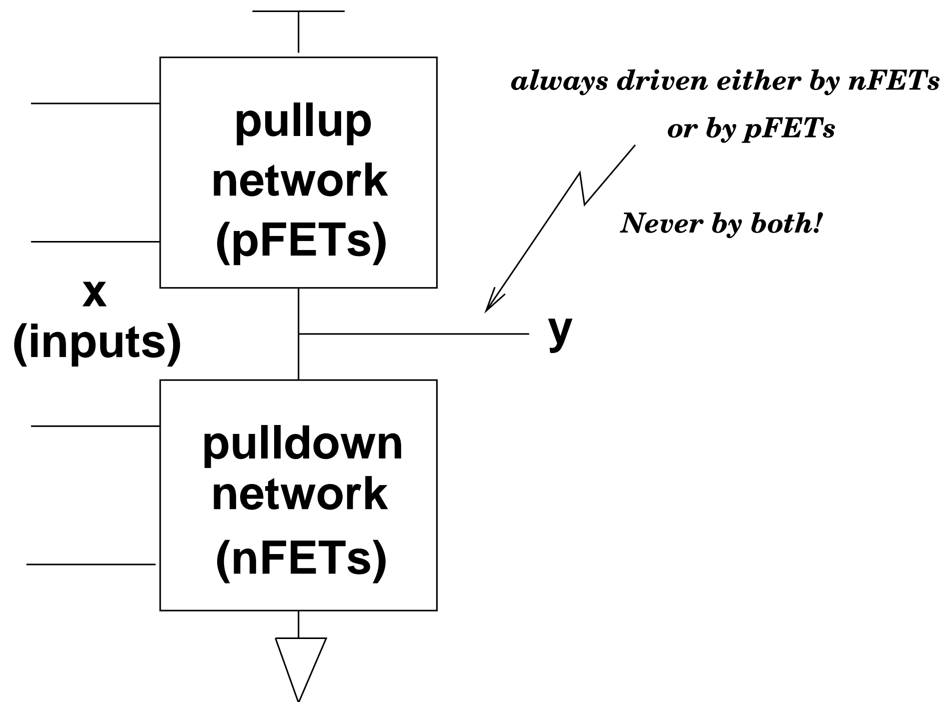
- Clocks
- Registers
- Maintaining value over time (Charge sharing, leaking)



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Where We Are

So far, we have spoken only of logic with no mention of *time:*




Analyze nFET network
Analyze pFET network

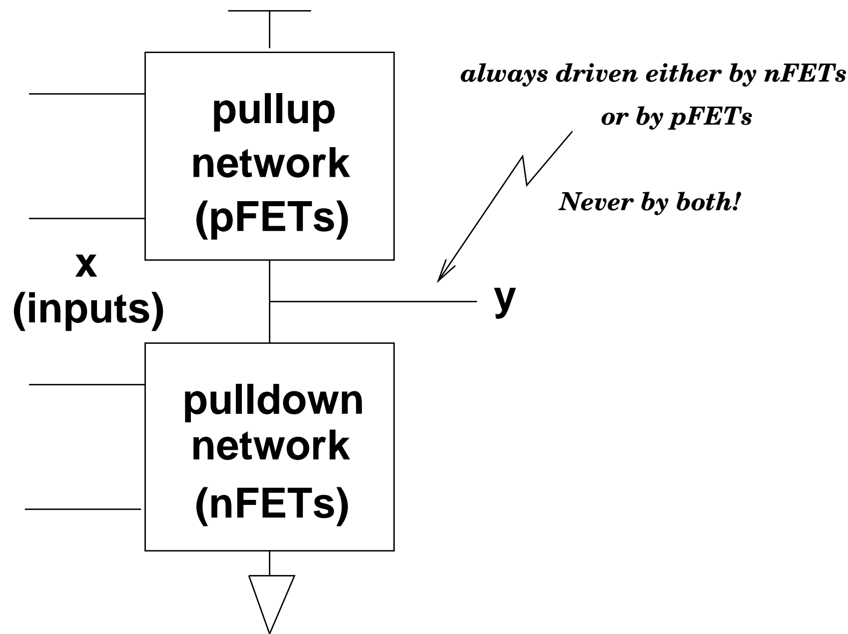
- Canonical sum-of-products
- K-maps
- Quine-McCluskey
- Gate sharing . . .



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# The Golden Rules

(for static CMOS design)




Four rules:

- Highs passed by pFETs.
- Lows passed by nFETs.
- Output always driven.
- No short circuits between the power supplies.

How (and when) do we change the outputs?



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# How Do We Design Large Systems?

- One big combinational circuit implementing a big function...? Impossible: size!
- Notion of *algorithm*: Sequence of steps, repetition trading size for execution time
- Notion of *pipelining*: Most algorithms (Systems) are used to compute a sequence of outputs for a sequence of inputs (with or without dependencies on each other)

$$...x3, x2, x1 \rightarrow [F] \rightarrow F(x3), F(x2), F(x1)...$$

Assume $F = k.h.g$. Pipelining is possible:

$$x4 \rightarrow [g] \rightarrow g(x3) \rightarrow [h] \rightarrow h.g(x2) \rightarrow [k] \rightarrow k.h.g(x1)$$

Throughput improved, latency worse!



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Sequencing

- In hardware, concurrency (doing things in parallel) is easier than sequencing...

$$y := f(x); z := g(y); w := h(z)$$

- When do we know that $f$ has finished computing $y$ and $g$ can use it to compute $z$? Two solutions:
- Use time: Clocked design (this term)
- don't use time...: asynchronous, clockless (next term)



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Using Clocks for Sequencing

$$y := f(x)$$

- We introduce clock signals $\phi$ and $\phi'$ such that $x$ is valid at $\phi$ and $y$ is valid at $\phi'$. We know and control $\delta$:

$$t(\phi') - t(\phi) = \delta + \epsilon$$

where $\epsilon$ is the *clock skew*.

- We choose $\delta$ such that the time it takes to compute $f(x)$ is less than $max(\delta + \epsilon)$ for all $x$
- Not only time constraint... More depending on choice of clock signals $\phi$ and $\phi'$.



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Two Basic Schemes

- Add the clock signal to the function: $y := f(x) \wedge \phi$
  precharge, dynamic, domino-style...

  (modern, for later...)
- Add the clock signal to the input:

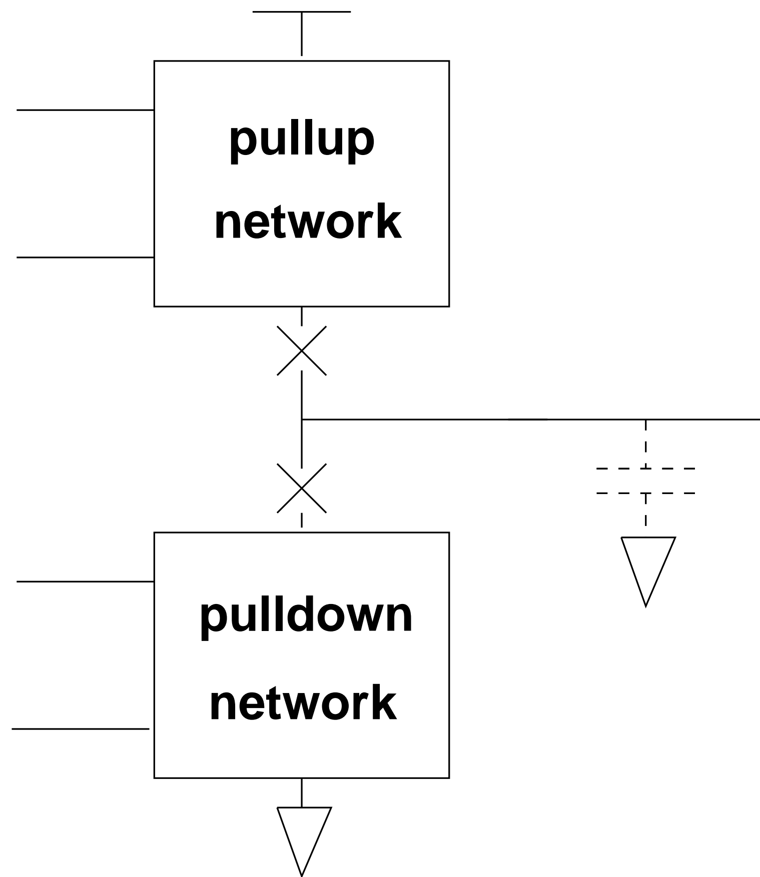
$$x' := x \wedge \phi \parallel y := f(x') \| y' := y \wedge \ \phi' \| \ z := g(y')...$$

Separate latching of inputs. (traditional, safe, for now...)



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# How do we keep floating values over time?




What happens? Recall

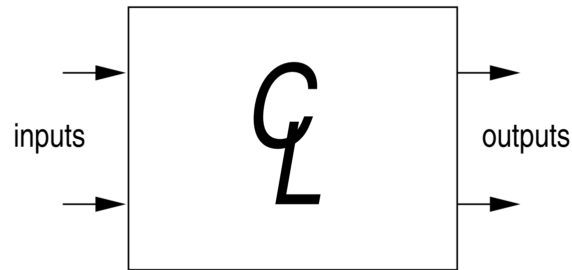
$$R_{\text{off}}/R_{\text{on}} \approx 10^5 \ldots$$

Thus: $\tau_{\text{off}}/\tau_{\text{on}} \approx 10^5$ as well. In other words, a *long* time. We will use this soon...



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Timing




Obviously, we need *some* notion of time so we can use the circuit repeatedly.

- Unambitious goal: Use combinational logic more than once. (As often as possible.)
- More ambitious goal: Trade circuit size for execution time.

This brings us to the two uses of clocks (or other means of synchronization):

- Circuit timing.
- Sequencing of algorithms.

*This is what keeps VLSI interesting.*



Copyright ©2010, California Institute of Technology. All Rights Reserved.

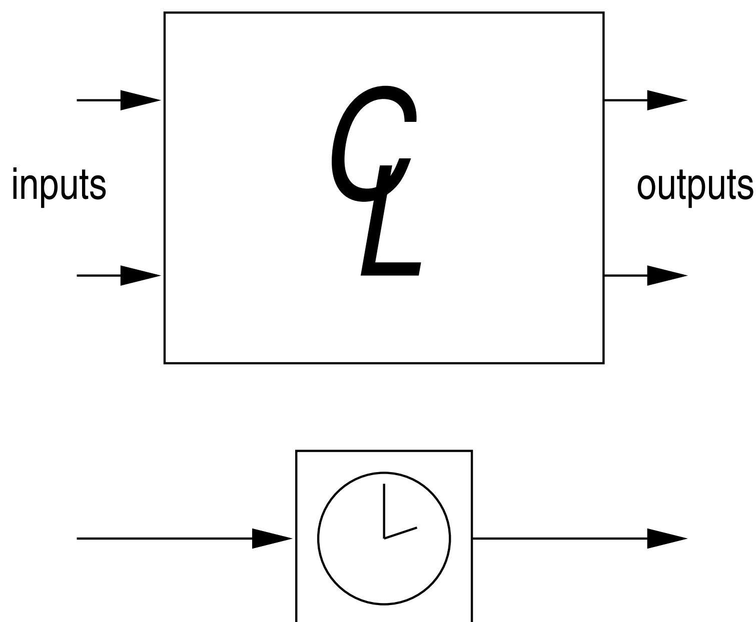
# Ways to approach the problem

To use a circuit repeatedly, we want to make sure that we give it enough time to compute its outputs.

General approach: Issue inputs to circuit, wait for it to be done computing, then look at the result.

How to accomplish the "wait for it to be done computing"?

Use a timer? (Asynchronous bundled data protocol.)



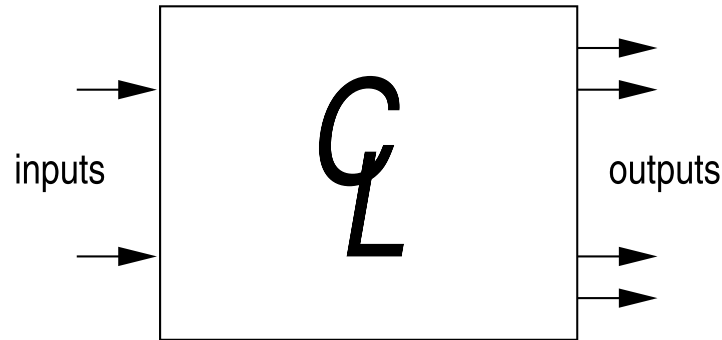

This works, but it's not used very much...



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Let the data do its own timing

We can make the "appearance" of output data signal that the computation is done.




output ⇒ validity

There are some subtle issues here:

- For starters, how do we know that the computation is done if the result is the same as the last time we used the circuit?
- Leads to handshake protocols. This happens for the bundled method too.
- No glitches allowed (at least on the outputs)!

See CS/EE 181 b, CS185 for more about this...



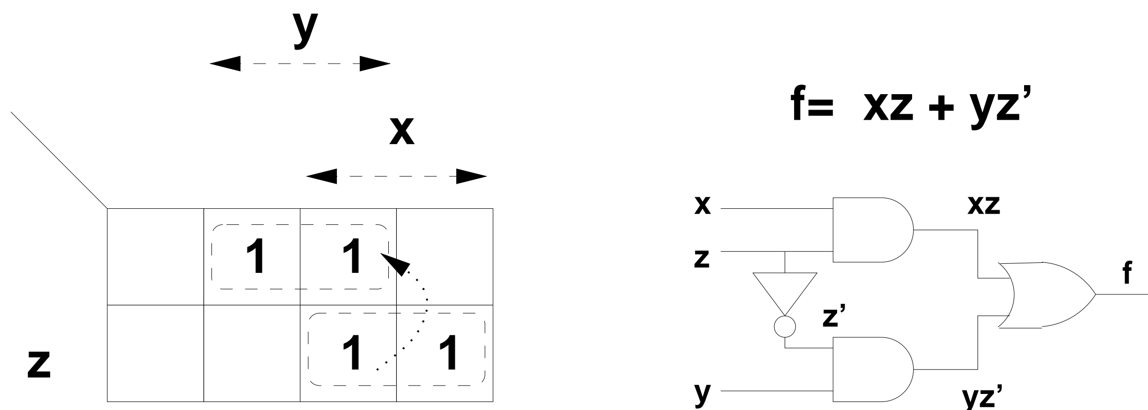
Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Another issue—glitches

Why is it hard to know when a circuit is “done”? Can we just wait for the output to change?

- Yes, but this requires a lot of care... (More than we are willing to invest right now.)

*Example.* Implement:




What happens when $z$ changes?

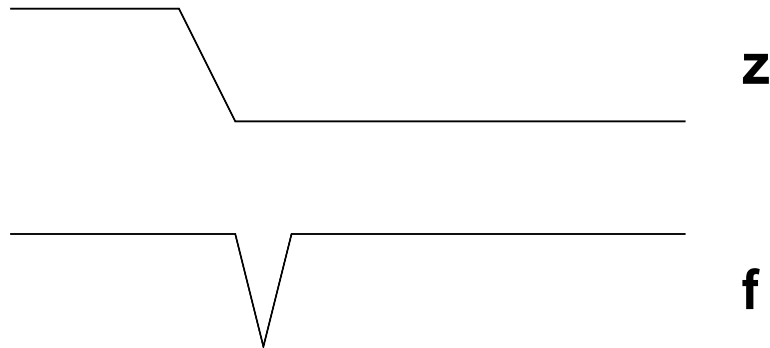
| time | $x$ | $y$ | $z$ | $w = z'$ | $f = xz + yw$ |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | 1 |



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# How to get rid of glitches

The $f = xz + yz'$ has a nasty glitch.




How can we get rid of this?

- In this particular case, we can cover an "extra" implicant. Makes circuit safe for any single input changing.
- Can't fix it for general input changes!

Two ways out:

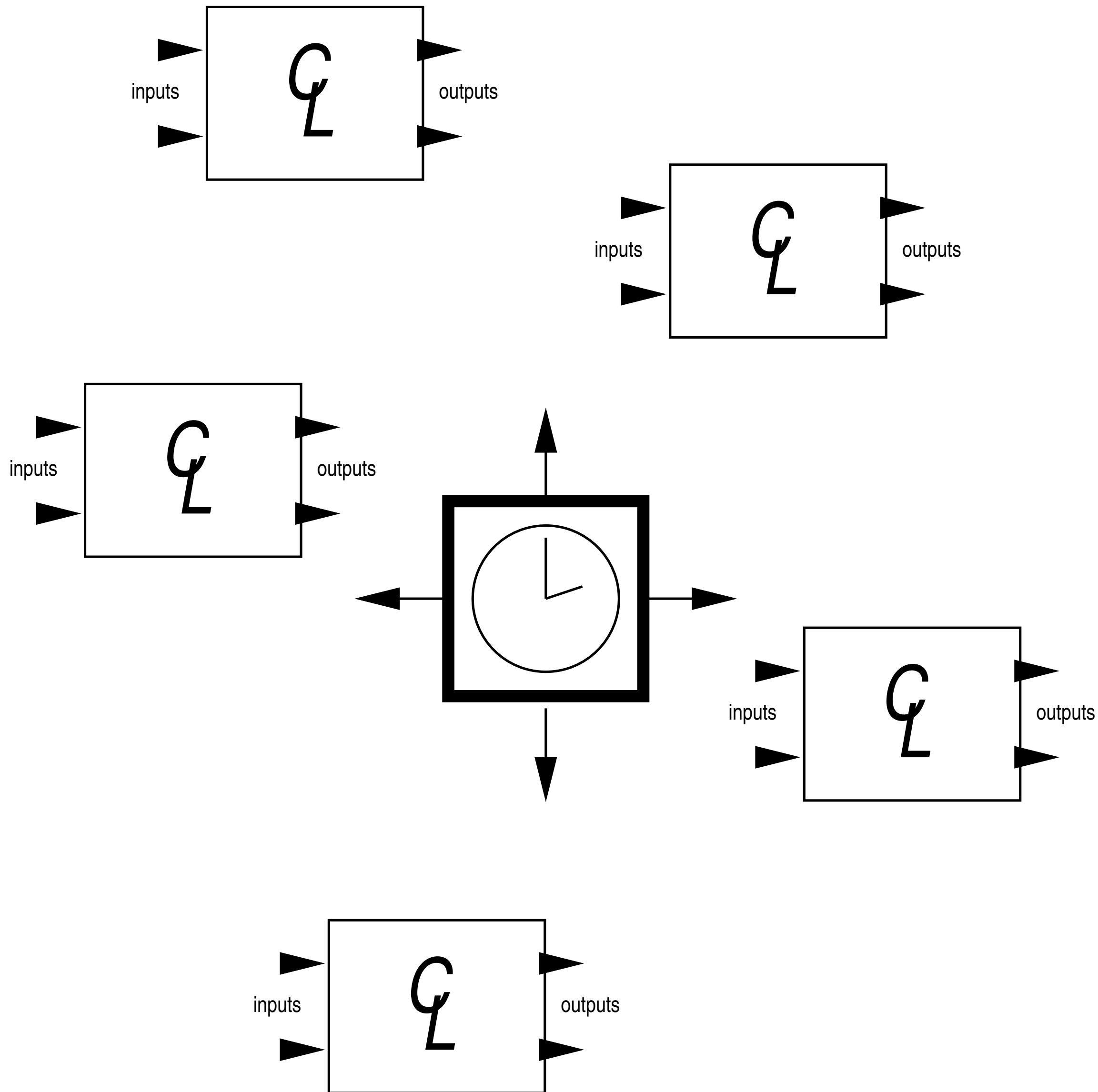
- Restricting our attention to classes of circuits that do not glitch. (This is a rather severe restriction—next term)
- Allowing the circuit to glitch, but ignoring it during that time. ("Wait until the outputs are ready.") Hopefully, it will eventually stabilize. . .
- Can be expensive in terms of energy in large combinational blocks. . .



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# For now—clocks

Standard method of solving the problem: Clocks.



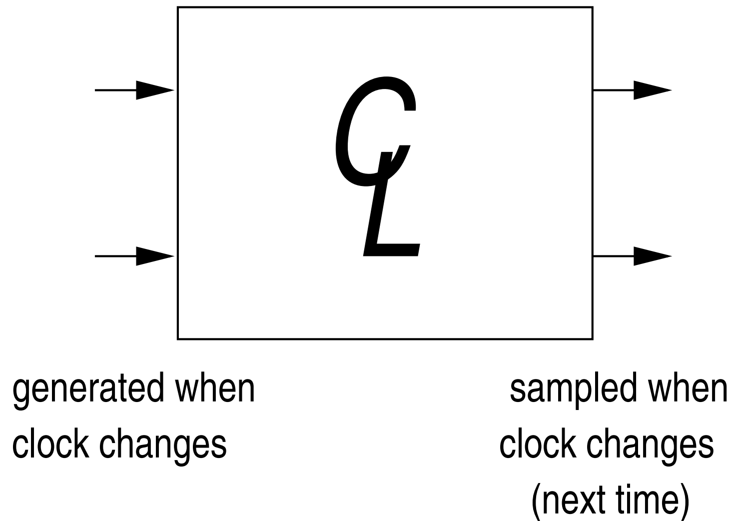

A single central *clock* keeps time for all the function units.



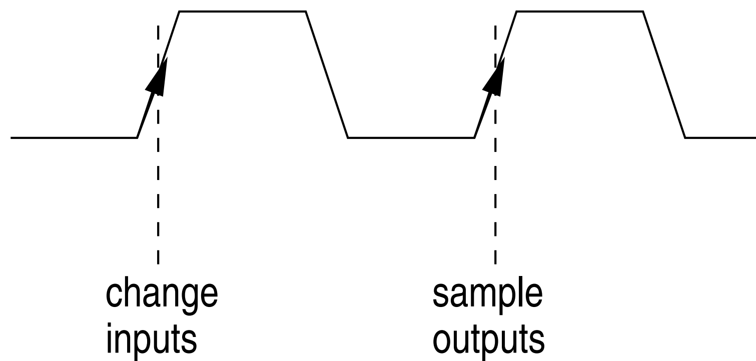
Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Clocking discipline

A simple clocking discipline. . .




"Edge-triggered" clocks (used a lot in TTL circuits).



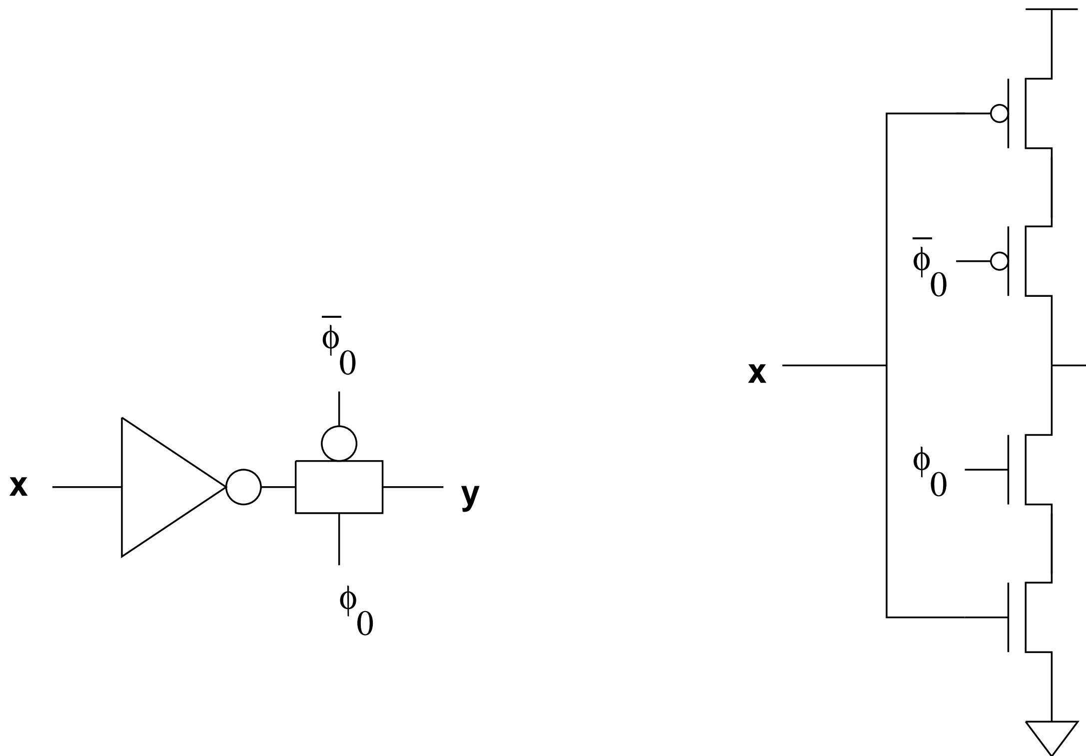

This exact approach not so good for CMOS, but it's on the right track.



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# A simple basic element

To choose our clocking approach, we need to examine the basic building blocks.




This behaves like a three-state D-type latch. . .
In CMOS, we can do this with very few transistors. Why? (For comparison: TTL D-type flip-flop has about 20-30 transistors. . .)
Output might float...

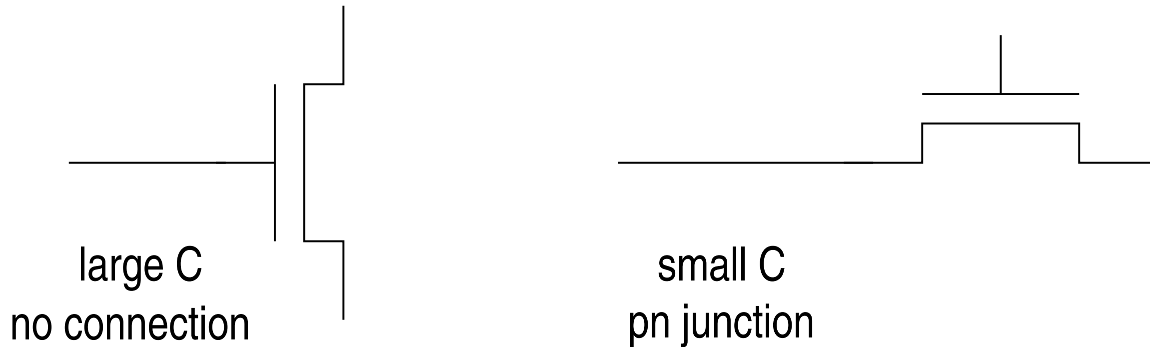


Copyright ©2010, California Institute of Technology. All Rights Reserved.

# When can we float?

But there are conditions...

- Wire must be a good capacitor to ground.




- We need a gate to store charge on.
- Avoid trying to store charge on nodes with larger and leakier sources and drains...



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Staticizer, also called Keeper

$$\begin{aligned} Bu &\rightarrow y\uparrow \\ Bd &\rightarrow y\downarrow \end{aligned}$$

- "Floating": no path ot Vdd or GND when $\neg Bu \wedge \neg Bd$
- What happens to the charges on the output node when the node is floating?
- **Add another path to Vdd and GND**

$$\begin{aligned} Bu \vee Cu &\rightarrow y\uparrow \\ Bd \vee Cd &\rightarrow y\downarrow \end{aligned}$$

- Simplest solution: $Cu = y$ and $Cd = \neg y$

$$\begin{aligned} Bu \vee \neg y_ &\rightarrow y\uparrow \\ Bd \vee y_ &\rightarrow y\downarrow \end{aligned}$$

with $y_ = \neg y$



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Standard Staticizer

- Problem: "Fight"
- Remember the requirement of non-interference between complementary PRS

$$
\begin{aligned}
Bu \vee \neg y_ &\rightarrow y\uparrow \\
Bd \vee y_ &\rightarrow y\downarrow
\end{aligned}
$$

- Now we can have $Bu$ and $y_$ or $Bd$ and $\neg y_$ true at the same time



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Fight in Staticizer

- Suppose we want to switch $y$ to xero
- $i1$ moves $V(y)$ towards VDD
- $i2$ moves $V(y)$ towards GND
- Strongest current wins the fight. We want $i2$ to win
- Make $i1$ very small by making pFET very resistive
- We say that the feedback inverter is "weak"

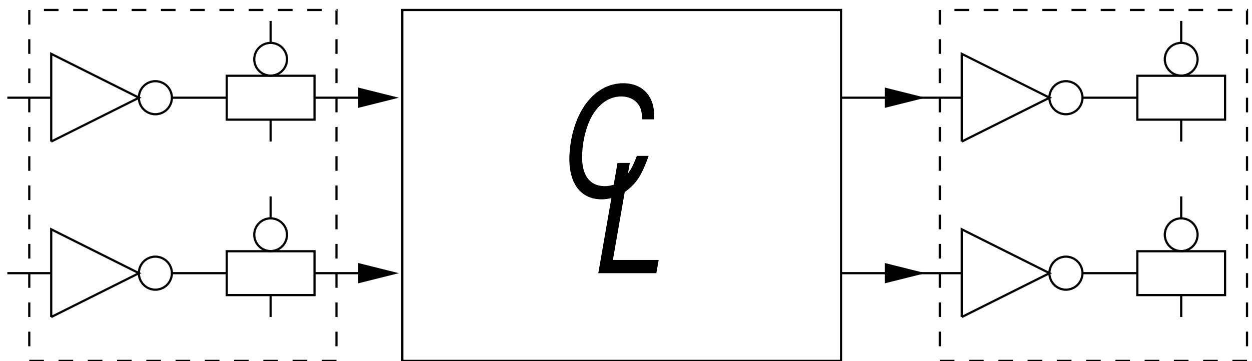


Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Basic clocking strategy

We design our combinational logic as in labs 1 and 2.

- Surround it with latches to control when we give it data and when we sample its outputs.




What do we use for the clocks?



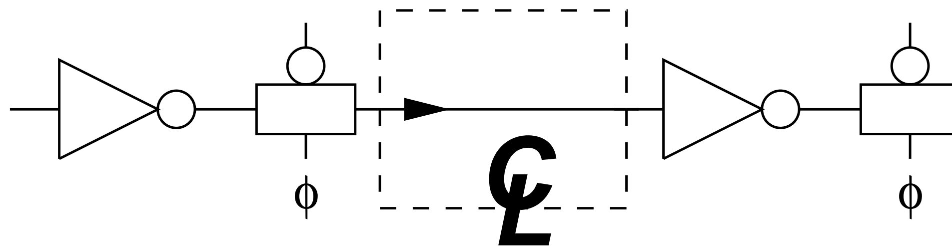
Copyright ©2010, California Institute of Technology. All Rights Reserved.

# "One-phase" clock

MOSFETS are level-sensitive. . .

We don't know how to build (simple) edge-triggered CMOS circuits!

⇒ One-phase clocking scheme is BAD.
Consider simplest "CL"— a wire:




Think of it as an air lock.

- If we open both valves at the same time, the signal can "race through" the system—it will show up a clock cycle early at the output, sending the rest of the system into confusion.



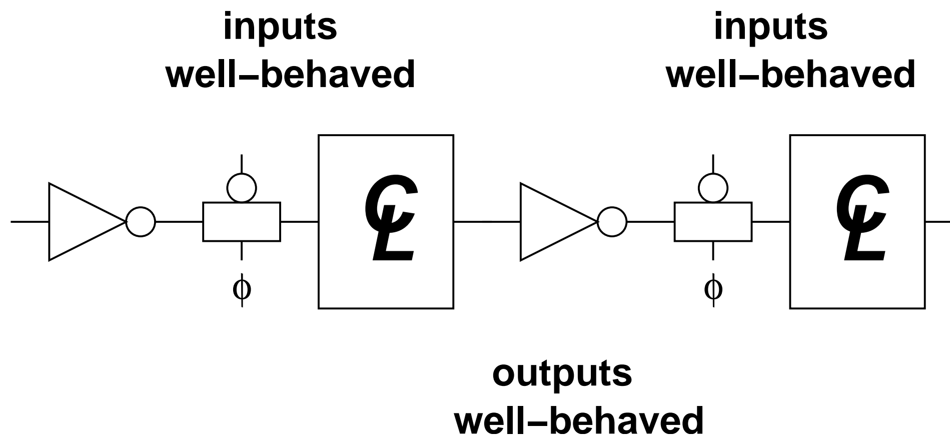
Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Clocking disciplines

The one-phase clock demands a strict *clocking discipline.*

E.g.,

- $CL$ must hold its value constant at the old value while the clock pulse is true.
- $CL$ must produce its new value by the time the second clock pulse comes around.




If I can guarantee this for each part of the circuit, I know it will work for the circuit as a whole!

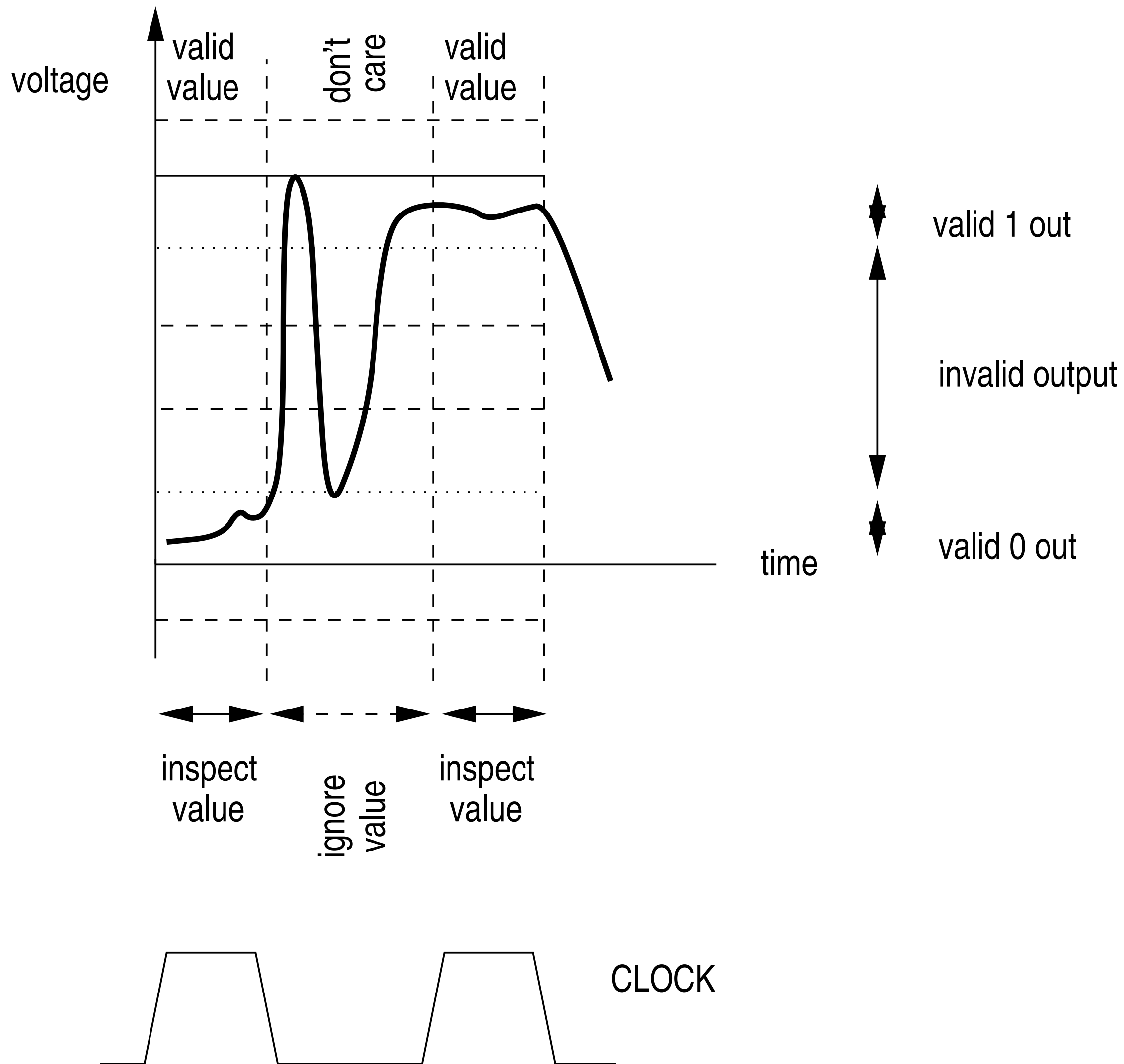


Copyright ©2010, California Institute of Technology. All Rights Reserved.

Doesn't this seem familiar?

- "Logic levels"?
- What do I mean by "done" anyhow?

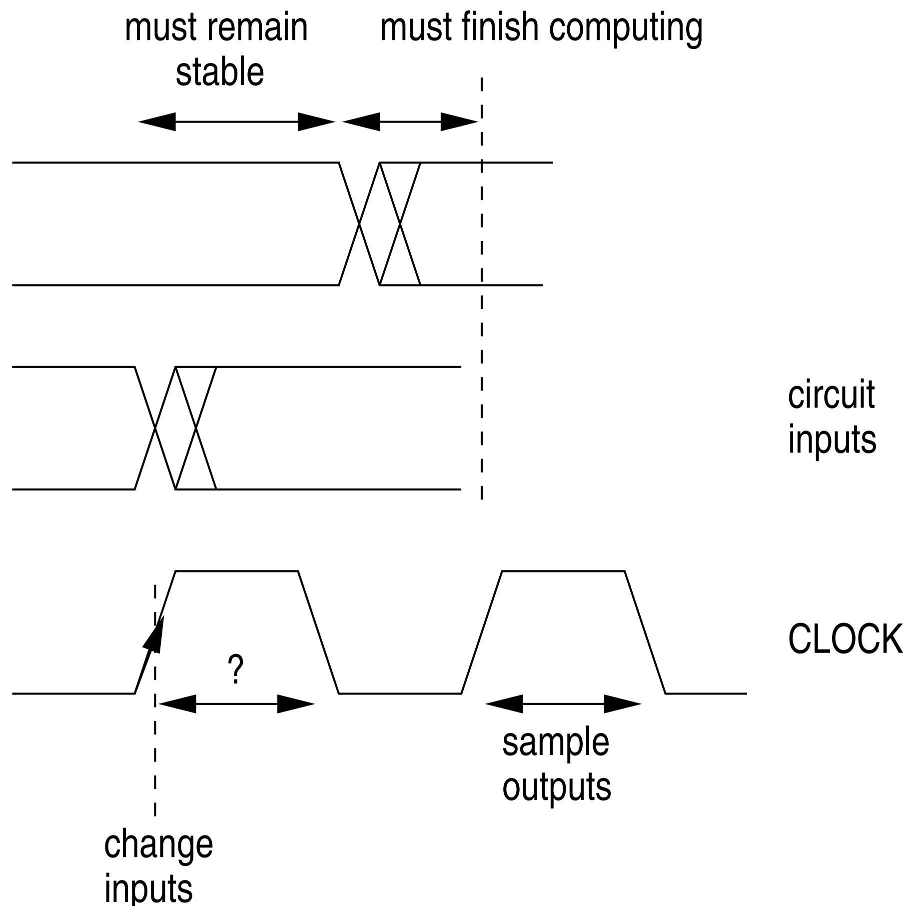
Actually it's the same thing as logic levels, except in time rather than in voltage. . .






Copyright ©2010, California Institute of Technology. All Rights Reserved.

# The problem—One-phase




- Too slow and we won't meet constraint on the right—won't be done computing when the clock samples the output.
- Too fast and we mess up the sampling of the previous value.

Two-sided constraint! *Very* difficult to deal with.
(But fast!)



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# “Two-phase” (nonoverlapping) clock

What we want is a timing discipline that is easier to obey.

- Sample inputs during one clock phase $\phi_0$.
- Produce outputs during one clock phase $\phi_1$.

$\phi_0$ and $\phi_1$ are not to overlap—there is no race.

- If it doesn't work because our circuits are too slow, just slow down the clock.
- Very “safe” and conservative.

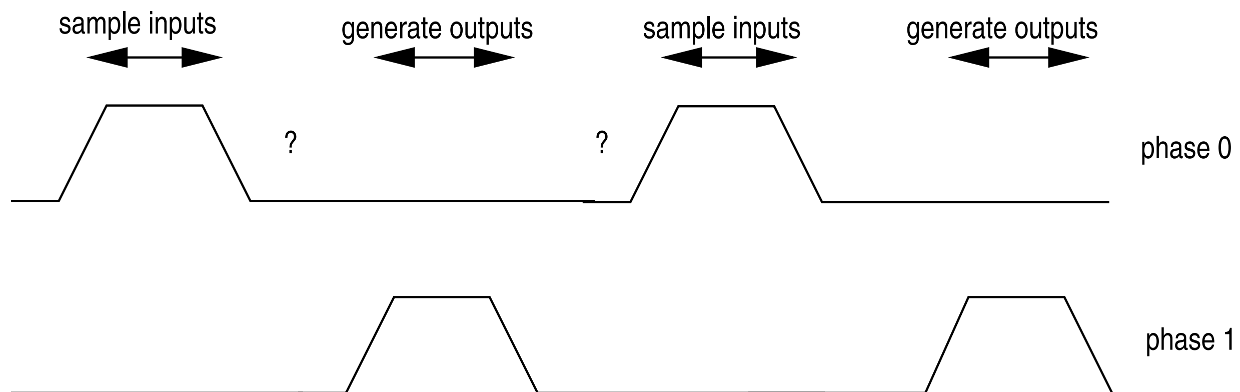


Copyright ©2010, California Institute of Technology. All Rights Reserved.

# A picture—Nonoverlapping clock

Two nonoverlapping phases.

- The gap between the phases takes care of clock *skew* in the system.




And when neither clock is asserted?

- State holding! (This is where the CMOS capacitors come in.)

Obvious disadvantages compared to one-phase clock:

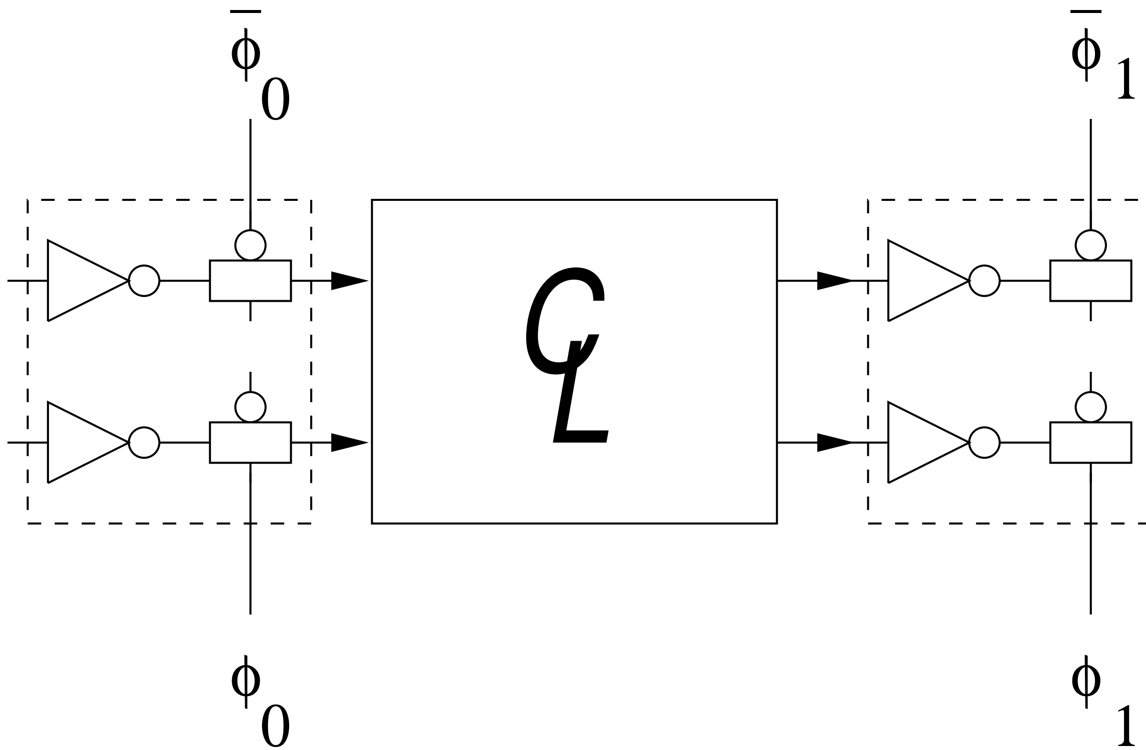
- Potentially lower performance.
- More clock wires.



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# How to do it

We will see this structure again and again for the rest of the term:



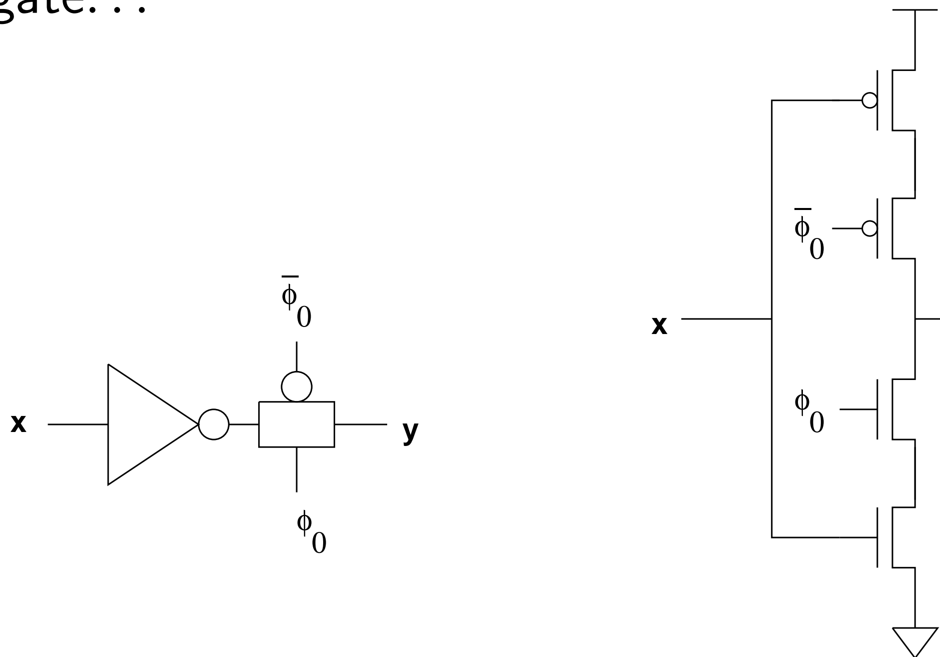

Of course, sometimes you may want to reverse the two.



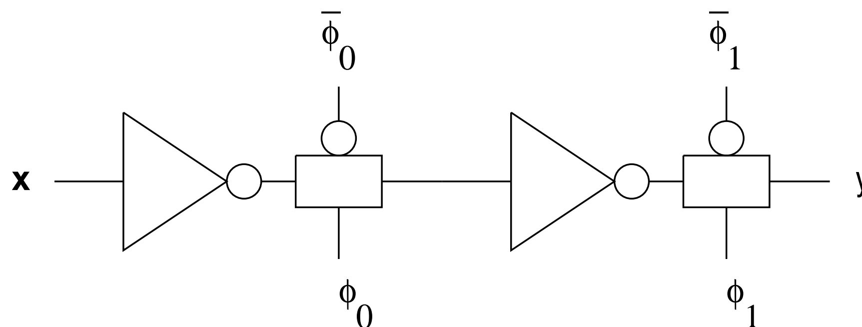
Copyright ©2010, California Institute of Technology. All Rights Reserved.

# In practice

We notice that we can disconnect the transistors in the pass gate. . .




We can call this a “half latch” or “half register”—why?




This is a *shift register.*




Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Maintaining the timing discipline

Now that we have set up our timing discipline, we need to make sure we can maintain it.

Two potential classes of problems:

- Input changes too soon. (Back to one-phase clock.)
- Output doesn't stay put during the 00 phase.

Can either of these happen?

Unfortunately, *yes.*

Case 1. Input changes too soon—*charge sharing.*

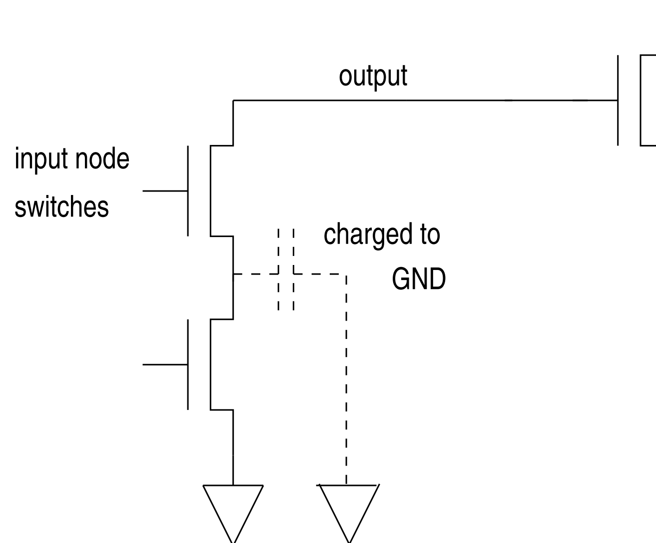
Case 2. Output doesn't stay put—*leakage.*



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Charge sharing

When we draw the circuit diagram with the transistors looking like capacitors, it is easy to forget the p-n junction capacitance around the source and drain.




- Up to approximately 1/4 the capacitance per unit area of gate.
- Especially bad (of course) if you use diffusion wires.
- Perimeter term—worst for small transistors.

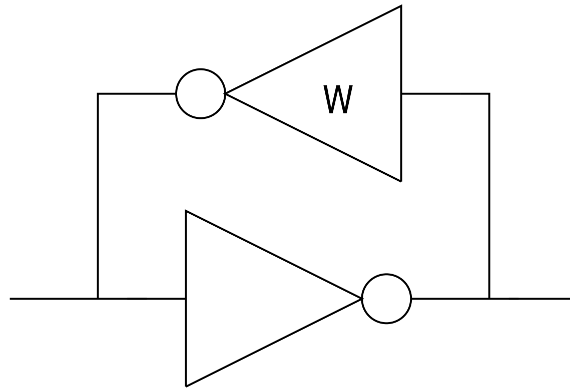
The internal node can act like a power supply on its own if there is nothing to keep the output high! Output may switch prematurely, defeating the clocking discipline.



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Staticizers

What if the capacitor leaks? What can we do about that?




(Example weak inverter: n-transistor $3\lambda$ wide (minimum size) by at least 12 long, p-transistor $3 \times 8$.) Use a state-holding element to hold the circuit's state. (In a sense, we are going back to the rule that the output always has to be driven.)

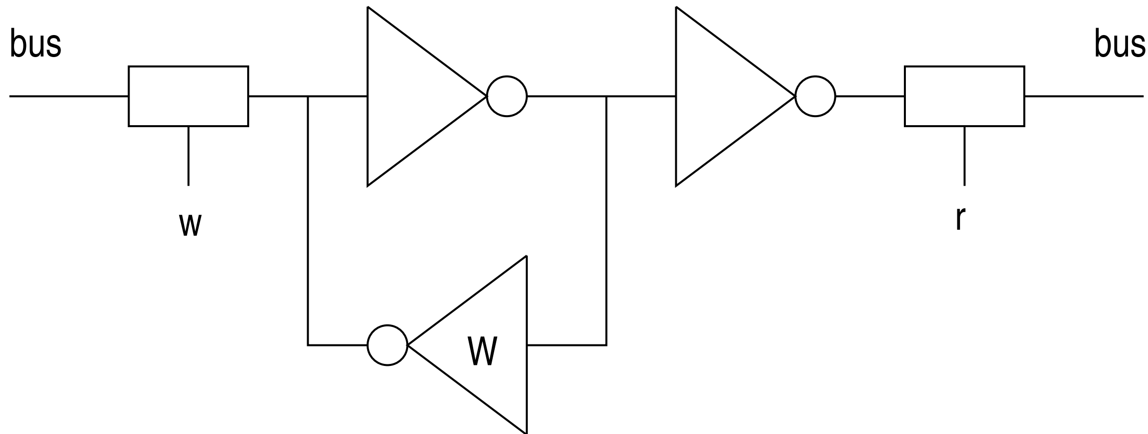
- Optional for nodes that are charged up on every clock cycle
- Important for easy testing.
- Essential for circuits that are not known to be charged every cycle.
- Essential for lab assignments!



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Sample register designs

For Lab 3, you will want to know how to design registers. (Not just shift registers.)



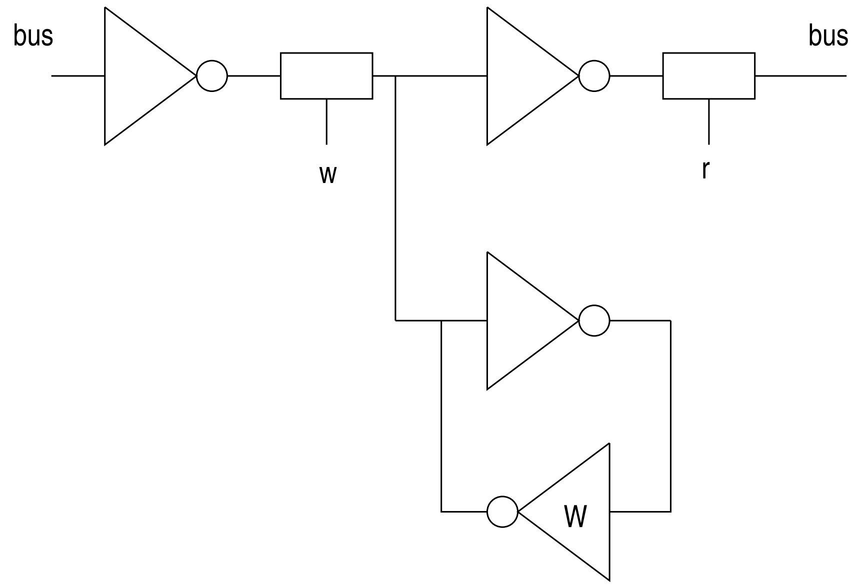

Normally, we enforce a clocking discipline for the bus and everything connected to it. Thus, $w = (\text{write}) \wedge \phi_0$ and $r = (\text{read}) \wedge \phi_1$. (The same discipline has to be obeyed by all units connected to the bus.)

- This register isn't perfect. . .

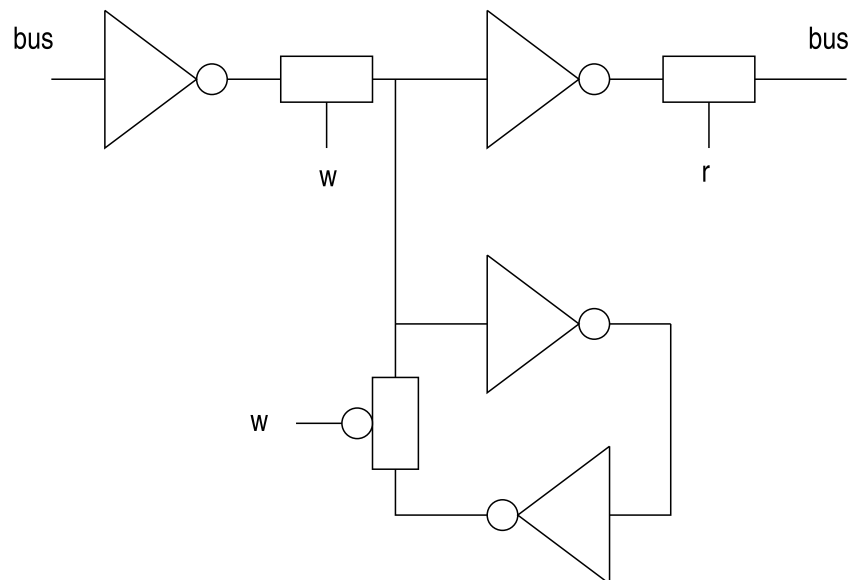


Copyright ©2010, California Institute of Technology. All Rights Reserved.

Two that work a bit better:




(12T)




(14T)

- How do we generalize the register to have multiple inputs and/or outputs? (Not difficult.)



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Register cookbook

Basic idea:

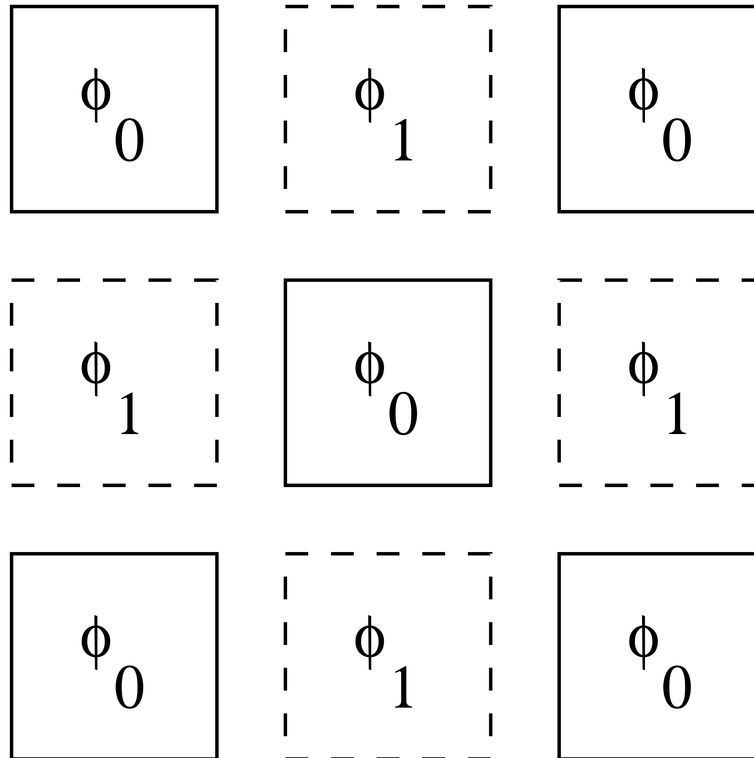
- Make sure that we drive the destination bus (won't be driven by it).
- Make sure that we are driven by source bus (won't drive it).
- Make sure all state-holding nodes are staticized.



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# The general idea

Imagine playing dominos:



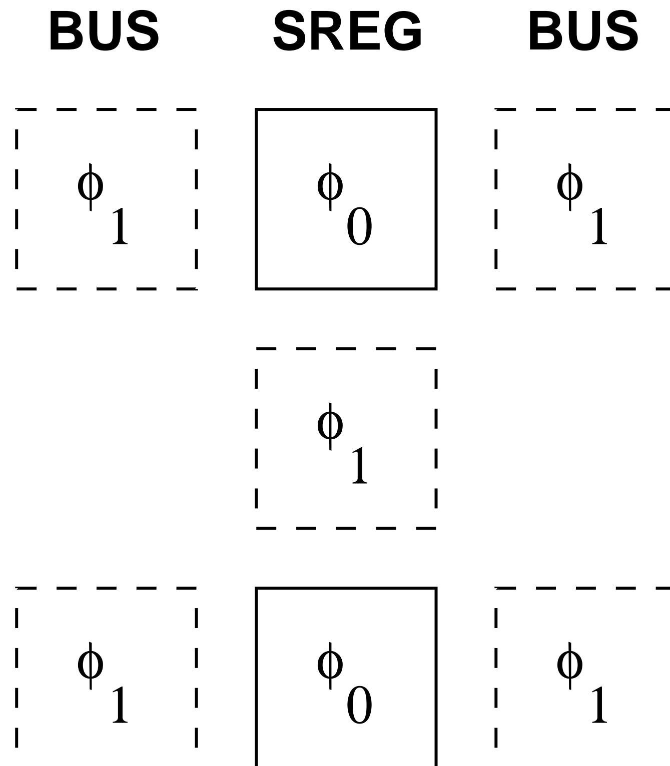

Solid cells clock on $\phi_0$, dashed cells on $\phi_1$, and only neighbors communicate.



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# A shift register

We can use this approach to see what we need for a shift register. . .




This is an official Lab 3 hint, but it works in general.

What happens if we do not have this alternating tiling?

Imagine the clock is *infinitely slow*—successive $\phi_0$ or $\phi_1$ latches are transparent at the same time!



Copyright ©2010, California Institute of Technology. All Rights Reserved.

# Next Time

Next time, we will continue talking about registers and how to use them together with logic.

- What to do about charge sharing.
- More register designs.

Other tricky applications of clocks in CMOS designs...

- Precharge logic.
- PLAs. (Special case of precharge logic.)

- Lab 3 examples...



Copyright ©2010, California Institute of Technology. All Rights Reserved.